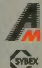
Programación del Z80

Rodnay Zaks



PR

PROGRAMACION DEL Z80

Programación del Z80

Rodnay Zaks



ANAYA MULTIMEDIA

INFORMATICA PERSONAL-PROFESIONAL

Título de la obra original:
PROGRAMMING THE Z80

Traducción de: Alfredo Cruz
Diseño de cubierta: Narcís Fernández

Villafranca, 22. 28028 Madrid
Depósito legal: M. 35.959-1985
I.S.B.N.: 84-7614-043-6
Printed in Spain
Imprime: Anzos, S. A. Fuenlabrada (Madrid)

Indice

Prólogo

Este libro es un texto completo para aprender a programar con el Z80, accesible a cualquiera aunque nunca haya escrito ningún programa, y útil, naturalmente, para quien trabaje con un Z80.

Quien ya sepa programar aprenderá aquí técnicas específicas que aprovechan las peculiaridades del Z80 o que derivan de ellas. El texto cubre las técnicas elementales e intermedias necesarias para empezar a programar.

La finalidad del libro es proporcionar un nivel realmente competente al lector que desee programar el microprocesador. Naturalmente, es imposible aprender a programar sólo con un libro, sin practicar, pero cabe esperar que el texto estimulará al lector hasta el punto de hacerle sentirse capaz de empezar a escribir programas y de resolver con un microordenador problemas de programación sencillos y hasta moderadamente complejos.

El libro parte de la experiencia del autor, que ha enseñado a programar microordenadores a más de mil alumnos, y por eso está altamente estructurado. Los capítulos van, por lo general, de lo simple a lo complejo. El lector que tenga ya conocimientos elementales de programación podrá saltarse el primer capítulo. Quienes, por el contrario, nunca hayan escrito un pro-

grama, quizá necesiten leer más de una vez las secciones finales de algunos capítulos. El texto llevará al estudiante a través de todos los conceptos y técnicas básicos necesarios para crear programas cada vez más complicados; por ello es muy recomendable que se respete el orden de los capítulos. Además, quien desee obtener de verdad resultados tangibles deberá esforzarse por resolver el mayor número posible de ejercicios. La dificultad de éstos se ha escalonado muy cuidadosamente, y todos están pensados para comprobar si el material propuesto se ha entendido por completo. Quien no realice los ejercicios no podrá aprovechar por completo el valor educativo de este libro. Varios de ellos, como el de multiplicación, resultarán laboriosos, pero al hacerlos se *aprende mediante la práctica*, que es la única manera de aprender a programar.

Para los que con este libro se aficionen a programar se está preparando otro titulado *Aplicaciones del Z80*, que le servirá de complemento.

En esta misma colección hay otros títulos que enseñan a programar otros microprocesadores diferentes.

Los verdaderamente interesados en el estudio del soporte físico deberían consultar los títulos *From Chips to Systems: an Introduction to Microprocessors* y *Microprocessor Interfacing Techniques*.

El contenido de este libro se ha verificado con la mayor atención, y puede considerarse de fiar, aunque, inevitablemente, se habrán deslizado errores tipográficos o de otra clase; a este respecto, el autor agradecerá cualquier observación que pueda ser útil para lectores de futuras ediciones. Se tendrán, igualmente, en consideración cualesquiera otras sugerencias de posibles mejoras, como programas deseados, desarrollados o considerados de valor por los lectores.

na vez las secciones
evará al estudiante a
ásicos necesarios para
dos; por ello es muy
los capítulos. Además.
dos tangibles deberá
posible de ejercicios.
uy cuidadosamente. y
el material propuesto
ealice los ejercicios no
ducativo de este libro.
ón. resultarán laborio-
la práctica. que es la

n a programar se está
Z80. que le servirá de

títulos que enseñan a
rentes.
el estudio del soporte
Chips to Systems: an
roprocessor Interfacing

rificado con la mayor
unque. inevitablemente.
o de otra clase; a este
observación que pueda
Se tendrán. igualmen-
sugerencias de posibles
arrollados o considera-



Con

Introducción

¿Qué [illegible] pro

1

Conceptos básicos

Introducción

En este capítulo presentaremos las ideas y definiciones básicas de programación de ordenadores. El lector familiarizado con estos temas quizá prefiera echar una ojeada rápida al contenido de estas páginas y pasar rápidamente al capítulo 2. Sin embargo, es aconsejable que incluso quien ya tenga experiencia lea esta introducción, porque en ella veremos conceptos muy importantes, como los de complemento a dos y representaciones BCD y de otro tipo. Algunos de ellos resultarán nuevos para el lector y, en cualquier caso, contribuirán a mejorar el nivel de conocimientos de los programadores con experiencia.

¿Qué es programar?

Ante un problema, lo primero que debe hacerse es idear una solución. A la expresión de ésta mediante una cadena de pasos sucesivos se le llama *algoritmo*. Un algoritmo es, pues, la especificación paso a paso de la solución de un problema. El

algoritmo debe terminar tras un número finito de pasos, y puede expresarse en cualquier lenguaje o conjunto simbólico. Un ejemplo de algoritmo sencillo sería el siguiente:

1. Meter la llave en la cerradura.
2. Dar a la llave una vuelta completa hacia la izquierda.
3. Agarrar el picaporte.
4. Girar el picaporte hacia la izquierda y empujar la puerta.

En este punto, si el algoritmo es el adecuado a la cerradura en cuestión, la puerta se abrirá. Esta serie de instrucciones en cuatro pasos constituye un algoritmo de apertura de una puerta.

Una vez que la solución a un problema se ha expresado en forma de algoritmo, éste debe ejecutarse en un ordenador. Por desgracia, es cosa sabida que los ordenadores no entienden español, ni cualquier otro lenguaje humano, debido a la *ambigüedad sintáctica* inherente a todos ellos. Lo único que el ordenador puede entender es un subconjunto claramente definido de un lenguaje humano, y a ese subconjunto se le llama *lenguaje de programación.*

La operación de transformar un algoritmo en una secuencia de instrucciones escritas en lenguaje de programación se llama *programar*. En términos estrictos, la traducción del algoritmo a lenguaje de programación debería llamarse *codificación*, puesto que la programación abarca también el diseño de los programas y las "estructuras de datos" que constituirán el algoritmo.

Para programar con eficacia hace falta no sólo conocer las técnicas de ejecución de algoritmos, sino también dominar todos los recursos que ofrece el soporte físico del ordenador —registros internos, memoria y dispositivos periféricos— y utilizar de forma creativa las estructuras de datos apropiadas. La descripción de estas técnicas será el objeto de los próximos capítulos.

La programación obliga también a observar una estricta disciplina de documentación para que los programas realizados por una persona puedan ser entendidos por otras (y por el autor, pasado cierto tiempo). La documentación ha de ser interna y externa al programa.

Se llama documentación interna al conjunto de comentarios incluidos en el propio cuerpo de un programa y que explican su funcionamiento.

Por documentación externa se entiende la serie de explicaciones escritas, manuales y diagramas de flujo escritos con independencia del programa.

Diagramas d

Figura 1.1
Diagrama de flujo de un s[...] ma encargado de mant[...] constante la temperatura [...] una habitación.

Diagramas de flujo

Entre la realización del *algoritmo* y la del *programa* se intercala casi siempre un *diagrama de flujo*, que no es sino una representación simbólica del algoritmo por medio de una secuencia de rectángulos y rombos que contienen los pasos del mismo. En los rectángulos se escriben las *órdenes* o "instrucciones ejecutables". Los rombos encierran *pruebas condicionales* del tipo "si la información X es cierta, emprender la acción A, y la B en caso contrario". La definición formal y la discusión de los diagramas de flujo se harán más adelante, al tratar de los programas.

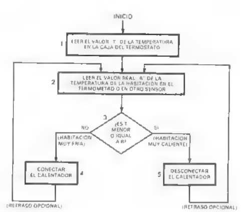


Figura 1.1
Diagrama de flujo de un sistema encargado de mantener constante la temperatura de una habitación.

En cualquier caso, el diagrama de flujo es un paso intermedio entre la especificación del algoritmo y la codificación muy recomendable. A este respecto, conviene señalar que se ha observado que aproximadamente el 10 por 100 de los programadores son capaces de escribir buenos programas sin recurrir al diagrama de flujo; ¡lo malo es que también se ha observado que el 90 por 100 de los programadores creen pertenecer a ese 10 por 100! Consecuencia, alrededor del 80 por 100 de los programas escritos por ese 90 por 100 fallan la primera vez que se pasan por el ordenador; como es natural, no se trata de porcentajes exactos. En concreto, son pocos los programadores noveles que comprenden la utilidad del diagrama de flujo y muchos los que escriben programas "sucios" o erróneos, lo que les obliga a perder mucho tiempo haciendo pruebas y corrigiendo (es lo que se llama *puesta a punto*). Por tanto, en todos los

casos es muy recomendable hacer un diagrama de flujo. Es una operación que lleva muy poco tiempo. pero que casi siempre propicia la redacción de un programa limpio que se ejecuta correcta y rápidamente. Hay programadores que, una vez dominada la técnica de trazado del diagrama de flujo, son capaces de visualizarlo mentalmente, sin necesidad de dibujarlo, aunque a costa de que los programas que escriben, al carecer de la documentación que constituye el propio diagrama de flujo, sólo resultan comprensibles para ellos. En resumen, siempre que se redacte un programa de cierta importancia, conviene acostumbrarse a la disciplina de diseñar el correspondiente diagrama de flujo. A lo largo de este libro veremos numerosos ejemplos de tales diagramas.

Representación de la información

Todos los ordenadores manipulan información organizada en forma de números o de caracteres. Pasaremos a continuación a examinar las formas de representación externa e interna de la información en el ordenador.

REPRESENTACION INTERNA DE LA INFORMACION

Toda la información contenida en un ordenador se almacena en forma de grupos de bits (*bit* es contracción de *dígito binario*, es decir, "0" o "1"). Las limitaciones de la electrónica convencional imponen una sola forma práctica de representación de la información: la lógica de dos estados, "0" y "1". Un circuito electrónico digital conoce habitualmente dos estados: conectado y desconectado, estados que corresponden a las representaciones lógicas "0" y "1". Dado que esos circuitos se utilizan para ejecutar funciones "lógicas", se les llama de "lógica binaria". Lo importante es que en la actualidad prácticamente todo el proceso de datos se lleva a cabo en formato binario. En el caso de los microprocesadores en general, y del Z80 en particular, los bits se organizan en grupos de ocho, conocidos por octetos y, más frecuentemente, por *bytes*. El conjunto de cuatro bits se llama *nibble*.

Veamos ahora de qué forma se representa internamente este formato binario. Dentro del ordenador hay que representar dos entidades: la primera es el programa o secuencia de instrucciones; la segunda es el conjunto de datos con los que trabajará el programa, que pueden ser de carácter numérico o alfanumérico. Examinaremos a continuación la representación del programa y de las dos categorías de datos mencionadas.

REPRESENTACION DEL PROGRAMA

Todas las instrucciones se representan internamente en forma de bytes o múltiplos de bytes. Lo que se llama "instrucción breve" queda representada por un solo byte, mientras que las instrucciones más largas ocupan dos o más. Como el Z80 es un microprocesador de ocho bits, extrae los bytes de la memoria uno tras otro, lo que significa que las instrucciones de un solo byte se ejecutan, en principio, más rápidamente que las de varios. Más adelante veremos la importancia que tiene este aspecto del juego de instrucciones de cualquier microprocesador, y en concreto del Z80, en el que se quiso proporcionar la mayor cantidad posible de instrucciones breves para optimizar la eficacia del programa. No obstante, la limitación de la longitud a ocho bits plantea restricciones notables, que expondremos en su momento. Este es un ejemplo clásico de compromiso entre velocidad y flexibilidad en programación. El código binario en el que se representan las instrucciones lo determina el fabricante. El Z80, como cualquier otro microprocesador, sale de fábrica provisto de un juego de instrucciones fijo. Dichas instrucciones, que determina el fabricante, se recogen al final del libro junto con su código. Los programas se expresan todos como secuencia de esas instrucciones binarias, que para el Z80 veremos en el capítulo 4.

REPRESENTACION DE DATOS NUMERICOS

Representar números no es operación sencilla, y es preciso diferenciar varios casos. Empezaremos por representar enteros, pasaremos a continuación a enteros con signo —positivos y negativos— y terminaremos con la representación de números decimales.

Para representar enteros puede recurrirse a la forma *binaria directa*, que no es sino la representación del valor decimal del número en sistema binario. En éste, el bit situado en el extremo derecho es igual a 2 elevado a la potencia 0: el situado a su izquierda equivale a 2 elevado a 1: el siguiente, a 2 elevado a 2, y el del extremo izquierdo vale 2 elevado a 7, igual a 128.

$$b_7b_6b_5b_4b_3b_2b_1b_0$$

representa

$$b_72^7 + b_62^6 + b_52^5 + b_42^4 + b_32^3 + b_22^2 + b_12^1 + b_02^0$$

Las potencias de 2 son:

$$2^7 = 128, 2^6 = 64, 2^5 = 32, 2^4 = 16, 2^3 = 8, 2^2 = 4, 2^1 = 2, 2^0 = 1$$

La representación binaria es análoga a la decimal. En ésta, "123" equivale a:

$$\begin{array}{rcrcr} & 1 \times & 100 & = & 100 \\ + & 2 \times & 10 & = & 20 \\ + & 3 \times & 1 & = & 3 \\ \hline & & & = & 123 \end{array}$$

Obsérvese que $100 = 10^2$, $10 = 10^1$, $1 = 10^0$.

En esta "notación posicional", cada cifra representa una potencia de 10. En el sistema binario, cada cifra binaria o bit representa una potencia de 2.

Ejemplo: en el sistema binario, "00001001" equivale a:

$$\begin{array}{lrcrl} & 1 \times & 1 & = 1 & (2^0) \\ & 0 \times & 2 & = 0 & (2^1) \\ & 0 \times & 4 & = 0 & (2^2) \\ & 1 \times & 8 & = 8 & (2^3) \\ & 0 \times & 16 & = 0 & (2^4) \\ & 0 \times & 32 & = 0 & (2^5) \\ & 0 \times & 64 & = 0 & (2^6) \\ & 0 \times & 128 & = 0 & (2^7) \\ \hline \text{en decimal:} & & & = 9 & \end{array}$$

Veamos algunos otros ejemplos: "10000001" equivale a:

$$\begin{array}{lrcr} & 1 \times & 1 = & 1 \\ & 0 \times & 2 = & 0 \\ & 0 \times & 4 = & 0 \\ & 0 \times & 8 = & 0 \\ & 0 \times & 16 = & 0 \\ & 0 \times & 32 = & 0 \\ & 0 \times & 64 = & 0 \\ & 1 \times & 128 = & 128 \\ \hline \text{en decimal:} & & = & 129 \end{array}$$

por tanto, "10000001" equivale al número decimal 129.

Al examinar la representación binaria de los números se entiende por qué los bits se numeran de 0 a 7 empezando por la derecha. El bit 0 es "b_0" y corresponde a 2^0; el bit 1 es "b_1" y corresponde a 2^1, y así sucesivamente.

La figura 1.2 recoge los binarios correspondientes a los números decimales comprendidos entre 0 y 255.

Figura 1.2
Tabla de equivalencias ent
sistemas decimal y binari

Decimal	*Binario*	*Decimal*	*Binario*
[illegible]	00000000	32	00100000
1	00000001	33	00100001
[illegible]	00000010	.	
3	00000011	.	
4	00000100	.	
5	00000101	63	00111111
6	00000110	64	01000000
7	00000111	65	01000001
8	00001000	.	
9	00001001	.	
10	00001010	127	01111111
11	00001011	128	10000000
12	00001100	129	10000001
13	00001101		
14	00001110	.	
15	00001111	.	
16	00010000		
17	00010001	.	
.			
.			
.		254	11111110
31	00011111	255	11111111

Figura 1.2
Tabla de equivalencias entre los sistemas decimal y binario.

Ejercicio 1.1: *¿Cuál es el valor decimal de "11111100"?*

Transformación de decimal a binario

Calcúlese el equivalente binario del número decimal "11":

$11 \div 2 = 5$ resto $1 \rightarrow 1$ (LSB)
$5 \div 2 = 2$ resto $1 \rightarrow 1$
$2 \div 2 = 1$ resto $0 \rightarrow 0$
$1 \div 2 = 0$ resto $1 \rightarrow 1$ (MSB)

El binario equivalente se obtiene leyendo la columna de la derecha desde abajo hacia arriba: 1011.

El equivalente binario de un decimal se calcula dividiéndolo sucesivamente por 2 hasta obtener un cociente nulo.

Ejercicio 1.2: *¿Cuál es el número binario equivalente al decimal 257?*

Ejercicio 1.3: *Transfórmese 19 a notación binaria, y de nuevo a decimal.*

Operaciones con datos binarios

Las reglas aritméticas son directas y sencillas. La suma, por ejemplo, es:

$$\begin{aligned} 0 + 0 &= \ 0 \\ 0 + 1 &= \ 1 \\ 1 + 0 &= \ 1 \\ 1 + 1 &= (1)\ 0 \end{aligned}$$

donde (1) significa el acarreo de 1 ("llevarse" 1: obsérvese que "10" en el sistema binario equivale al decimal "2"). La sustracción binaria se realiza sumando el complemento, y se explicará al tratar de la representación de números negativos.

Ejemplo:

$$\begin{array}{rr} (2) & 10 \\ +\ (1) & +\ 01 \\ \hline =\ (3) & 11 \end{array}$$

La suma se realiza igual que en base decimal, sumando las columnas una por una, a partir de la primera por la derecha:

Suma de la columna derecha:

$$\begin{array}{r} 10 \\ +\ 01 \\ \hline 1 \end{array}$$

(0 + 1 = 1. No se lleva nada)

Suma de la siguiente columna:

$$\begin{array}{r} 10 \\ +\ 01 \\ \hline 11 \end{array}$$

– (1 + 0 = 1. No se lleva nada)

Ejercicio 1.4: *Calcúlese 5 + 10 en el sistema binario, comprobando si el resultado es efectivamente 15.*

Algunos otros ejemplos de adición binaria:

$$\begin{array}{rr} 0010 & (2) \\ +\ 0001 & (1) \\ =\ 0011 & (3) \end{array} \qquad \begin{array}{rr} 0011 & (3) \\ +\ 0001 & (1) \\ \hline =\ 0100 & (4) \end{array}$$

Este último ejemplo ilustra la función del acarreo.

En efecto, fijémonos en la columna de la derecha: 1 + 1 = (1)0; una vez efectuada la suma, llevamos 1, que se suma a la siguiente columna:

001	la columna 0 acaba de sumarse
+ 000	
+ 1	(acarreo)
= (1)0	siendo (1) el nuevo acarreo a la columna 2.

El resultado final es 0100.
Otro ejemplo:

0111	(7)
+ 0011	+ (3)
1010	= (10)

En este ejemplo vuelve a generarse un acarreo, que se lleva hasta la columna de la izquierda.

Ejercicio 1.5: *Calcúlese la suma:*

1111
+0001
= ?

¿Cabe el resultado en cuatro bits?

Por tanto, con ocho bits pueden representarse directamente los números comprendidos entre "00000000" y "11111111", es decir, entre "0" y "255". Inmediatamente se plantean dos dificultades: primera, que sólo representamos números positivos; segunda, que la magnitud de esos números queda limitada a 255, si trabajamos con sólo ocho bits. Veamos de qué forma se resuelven.

Binario con signo

En un número representado en notación binaria con signo, éste viene indicado por el bit de la izquierda, tradicionalmente "0", si es *positivo*, y "1", si es *negativo*; por tanto, "11111111" representa − 127, y "01111111", + 127. De esta forma ya podemos representar números positivos y negativos, pero a costa de reducir la magnitud de 255 a 127.

Ejemplo: "0000 0001" equivale a + 1 (el primer "0" es "+"; el resto, "000 0001", es igual a 1).

"1000 0001" = − 1 (el primer "1" es "−").

Ejercicio 1.6: *¿Cómo se representaría "− 5" en notación binaria con signo?*

Pasemos ahora al problema de la *magnitud*. Para representar números más grandes no hay más remedio que utilizar mayor número de bits. Así, con dieciséis bits (dos bytes) podemos representar todos los números comprendidos entre − 32K y + 32K (en lenguaje informático, 1K es igual a 1 024). El bit 15 lleva el signo, y los quince restantes (los comprendidos entre el 0 y el 14) expresan la magnitud: $2^{15} = 32K$. Si esta magnitud sigue siendo pequeña, no hay más que usar tres bytes o más. En resumen, cuanto más grande sea la magnitud del entero que queramos representar, tanto mayor será el número de bytes necesario para ello. Por eso, las versiones más sencillas del BASIC y otros lenguajes disponen de una precisión limitada para representar enteros, ya que necesitan manipular internamente las cantidades en un formato más corto. Las mejores versiones del BASIC y de los demás lenguajes ofrecen más cifras decimales significativas, pero a costa de reservar más bytes para cada número.

Otro extremo que debemos considerar es el de la velocidad. Veamos, por ejemplo, cómo se lleva a cabo la adición de dos números en la representación binaria con signo que acabamos de estudiar. Sea la suma de "− 5" y "+ 7":

+ 7 se representa como 00000111
− 5 se representa como 10000101

la suma binaria es 10001100, o − 12

Pero el resultado correcto no es − 12, sino + 2. Para trabajar en esta representación hay que atenerse a ciertas reglas determinadas, que dependen del signo. La consecuencia es que aumenta la complejidad y disminuye la eficacia. En otras palabras, la adición binaria de números con signo "no marcha bien". La situación es, por tanto, delicada, porque el ordenador, además de representar información, tiene que ejecutar con ella operaciones aritméticas.

La solución al problema viene dada por lo que se llama representación en *complemento a dos*, que sustituye a la *binaria con signo*. En lugar de abordarla directamente, nos detendremos antes un poco en el *complemento a uno*.

Complemento a uno

En complemento a uno todos los enteros positivos se representan en su formato binario correcto. Así, "+ 3" se representa como 00000011; por el contrario, "− 3" se representa determinando el complemento de cada uno de los bits de la representación original, lo que equivale a transformar todos los 0 en 1 y todos los 1 en 0. En el ejemplo que nos ocupa, la representación de "− 3" en complemento a uno sería 11111100.

Otro ejemplo:

+ 2 es 00000010
− 2 es 11111101

Obsérvese que en esta representación el primer bit de la izquierda es "0", si el número es positivo, y "1", si es negativo.

Ejercicio 1.7: *La representación de "+ 6" es 00000110". ¿Cuál será la de "− 6" en complemento a uno?*

Probemos a sumar ahora − 4 y + 6:

```
              − 4 es 11111011
              + 6 es 00000110
la suma es:   (1)    00000001, donde (1) indica un acarreo.
```

El "resultado correcto" será, por tanto, "2", o bien "00000010".

Veamos otro caso:

```
              − 3 es 11111100
              − 2 es 11111101
la suma es:   (1)    11111001
```

a bien "− 6" más una unidad que se acarrea; pero, en realidad, debería ser "− 5", es decir, 11111010, lo que quiere decir que el procedimiento no funciona.

Este formato sirve para representar números positivos y negativos, pero el resultado de la adición no siempre es correcto. Es preciso, por tanto, idear otra representación, que en este caso será el complemento a dos, evolución del complemento a uno.

Representación en complemento a dos

Los números positivos se representan como binarios con signo, exactamente igual que se hacía en complemento a uno.

La diferencia estriba en la representación de los *números negativos*, que se hace determinando primero el complemento a uno y *sumando uno* a continuación.

Veámoslo en un ejemplo: + 3 se representa como binario con signo por 00000011; su representación en complemento a uno es 11111100. El complemento a dos se obtiene sumando 1 a esta última, lo que da 11111101.

Apliquemos el procedimiento a la adición:

(3)	00000011
+ (5)	+ 00000101
= (8)	= 00001000

el resultado es correcto.

Veamos ahora qué ocurre en la sustracción:

(3)	00000011
(− 5)	+ 11111011
	= 11111110

Para identificar el resultado, calcularemos el complemento a dos:

el complemento a dos de 11111110 es 00000001
se suma 1 + 1
el complemento a dos es, pues 00000010, o + 2

El resultado anterior —"11111110" representa "−2" y es, por tanto, correcto.

Los resultados —ignorando el arrastre— han sido correctos tanto en la adición como en la sustracción, lo que parece indicar que el complemento a dos funciona.

Ejercicio 1.8: *¿Cuál es la representación de "+ 127" en complemento a dos?*

Ejercicio 1.9: *¿Cuál es la representación de "− 128" en complemento a dos?*

Probemos ahora a sumar + 4 y − 3 (la sustracción se realiza sumando el complemento a dos):

+ 4 es 00000100
− 3 es 11111101
el resultado es: (1) 00000001

Si ignoramos el acarreo, el resultado es 00000001, es decir, "1" en representación decimal, es, por tanto, correcto. Aunque no daremos la demostración matemática completa, digamos por el momento que esta representación funciona y que en complemento a dos se pueden sumar y restar números con signo con independencia del mismo. Al aplicar la regla normal de la adición en binario se obtiene el resultado correcto incluyendo el signo. El arrastre se ignora, lo que constituye una ventaja considerable, porque en caso contrario sería preciso corregir siempre el signo en el resultado, con el consiguiente alargamiento del tiempo de operación.

Para resumir, digamos que el complemento a dos constituye la forma de representación más adecuada para los procesadores más simples, como los microprocesadores. En procesadores complejos puede recurrirse a otras formas de representación, como el complemento a uno, usando un circuito especial para corregir el resultado.

A partir de este punto, todos los enteros con signo se supondrán representados internamente en notación de complemento a dos. La tabla de la figura 1.3 recoge los complementos a dos de varios números.

Ejercicio 1.10: *¿Cuáles son los números máximo y mínimo que pueden representarse en complemento a dos con sólo un byte?*

Ejercicio 1.11: *Calcúlese el complemento a dos de 20 y a continuación el del resultado obtenido. ¿Se vuelve a obtener 20?*

Veamos a continuación, mediante algunos ejemplos, la forma en que se aplica el complemento a dos. Llamaremos C al acarreo, que corresponde al bit 8 del resultado.

V indica desbordamiento o cambio de signo "accidental" a consecuencia de la excesiva magnitud de los números con que se opera. Se trata básicamente de un acarreo interno del bit 6 al bit 7 (el bit de signo), y examinaremos sus consecuencias a continuación.

Acarreo C

He aquí un ejemplo de acarreo:

```
    (128)          10000000
  + (129)        + 10000001
  -------        ----------
    (257) = (1)    00000001
```

siendo (1) el acarreo.

+	*Código en complemento a dos*	–	*Código en complemento a dos*
+127	01111111	–128	10000000
+126	01111110	–127	10000001
+125	01111101	–126	10000010
. . .		–125	10000011
		. . .	
+65	01000001	–65	10111111
+64	01000000	–64	11000000
+63	00111111	–63	11000001
. . .		. . .	
+33	00100001	–33	11011111
+32	00100000	–32	11100000
+31	00011111	–31	11100001
. . .		. . .	
+17	00010001	–17	11101111
+16	00010000	–16	11110000
+15	00001111	–15	11110001
+14	00001110	–14	11110010
+13	00001101	–13	11110011
+12	00001100	–12	11110100
+11	00001011	–11	11110101
+10	00001010	–10	11110110
+9	00001001	–9	11110111
+8	00001000	–8	11111000
+7	00000111	–7	11111001
+6	00000110	–6	11111010
+5	00000101	–5	11111011
+4	00000100	–4	11111100
+3	00000011	–3	11111101
+2	00000010	–2	11111110
+1	00000001	–1	11111111
+0	00000000		

Figura 1.3
Tabla de complementos a dos.

El resultado exige el uso de un noveno bit o bit 8 (ya que la cuenta empieza por el bit 0), llamado bit de arrastre.

Si suponemos que el acarreo es el noveno bit del resultado, vemos que éste es, efectivamente, 100000001 = 257.

El acarreo debe detectarse y manipularse con atención. En el interior del microprocesador, los registros encargados de almacenar la información tienen, por lo general, una amplitud de sólo ocho bits; en este ejemplo sólo se registrarían los bits 0 a 7.

Código en complemento a dos
10000000
10000001
10000010
10000011
10111111
11000000
11000001
11011111
11100000
11100001
11101111
11110000
11110001
11110010
11110011
11110100
11110101
11110110
11110111
11111000
11111001
11111010
11111011
11111100
11111101
11111110
11111111

bit o bit 8 (ya que la
de arrastre.
no bit del resultado,
01 = 257.
se con atención. En
tros encargados de
neral, una amplitud
registrarían los bits

Por tanto, el acarreo exige siempre un cuidado especial, y debe detectarse y procesarse mediante instrucciones determinadas. El proceso sigue una de estas tres alternativas: almacenar el acarreo en algún otro sitio (mediante una instrucción especial), ignorarlo o, si el mayor resultado permitido es "11111111", considerarlo un error.

Desbordamiento O

Veamos un ejemplo de desbordamiento:

```
bit 6──┐
bit 7──┐│
       ▼▼
     01000000       (64)
   + 01000001     + (65)
   = 10000001   = (− 127)
```

Como se observa, se ha producido un acarreo interno del bit 6 al bit 7; es lo que se llama desbordamiento.

Debido a ese "accidente", el resultado es negativo; es, pues, preciso detectar la situación, para corregirla.

Examinemos otro caso:

```
         11111111     (− 1)
       + 11111111   + (− 1)
   = (1) 11111110   = (− 2)
      ▼
   acarreo
```

En este caso se ha producido un acarreo interno del bit 6 al bit 7, y de éste al bit 8 (el acarreo C que analizamos en la sección anterior). Como las reglas del complemento a dos especifican que dicho acarreo debe ignorarse, el resultado obtenido es el correcto.

Estrictamente hablando, esta última no es una situación de desbordamiento, ya que el acarreo del bit 6 al bit 7 no ha tenido como consecuencia el cambio de signo.

Al trabajar con números negativos, el desbordamiento no se limita a un acarreo del bit 6 al bit 7.

Veamos un nuevo ejemplo:

```
         11000000    (− 64)
       + 10111111    (− 65)
   = (1) 01111111   (+ 127)
      ▼
   acarreo
```

Esta vez no ha habido acarreo interno del bit 6 al bit 7, sino acarreo externo, pero el resultado es, no obstante, incorrecto, porque ha cambiado el bit 7: se trata de una situación de desbordamiento.

En resumen, se producirá desbordamiento en cuatro casos:

1. Adición de números positivos muy grandes.
2. Adición de números negativos muy grandes.
3. Sustracción de un positivo muy grande a un negativo muy grande.
4. Sustracción de un negativo muy grande ■ un positivo muy grande.

Vamos a tratar de mejorar la anterior definición de desbordamiento.

Desde un punto de vista técnico, se reserva un bit especial, o indicador de desbordamiento, llamado "bandera" para cuando se produzca arrastre del bit 6 al bit 7, sin que haya acarreo externo, o cuando no haya arrastre del bit 6 al bit 7, pero sí acarreo externo. Esto significa que el bit 7, y, por tanto, el signo del número, ha cambiado accidentalmente. Para el lector interesado por los aspectos técnicos, diremos que la bandera de desbordamiento se determina sometiendo a la operación O exclusiva los acarreos que llegan al bit 7 y los que salen de él. Prácticamente, todos los microprocesadores disponen de una bandera especial de desbordamiento que detecta automáticamente esa situación para que pueda emprenderse la acción correctora.

La presencia de desbordamiento significa que el resultado de la suma o la resta exige más bits que los disponibles en el registro usual de ocho bits utilizado para almacenarlo.

El acarreo y el desbordamiento

Los bits de acarreo y desbordamiento se llaman "banderas". Existen en todos los microprocesadores, y en el próximo capítulo aprenderemos a aprovecharlos para crear programas efectivos. Ambos se encuentran en un registro especial llamado de banderas ■ de "estado", que contiene, además, otros indicadores, cuya función se abordará en el capítulo 4.

Ejemplos

Veamos a continuación el comportamiento del acarreo y el desbordamiento mediante algunos ejemplos prácticos. En todos los casos, el símbolo O denotará el desbordamiento, y el C, el acarreo.

Si no hay desbordamiento O = 0, y en caso contrario, O = 1; lo mismo para el acarreo C. Recuerde que las reglas de aplicación del complemento a dos exigen que se ignore el acarreo (aunque no daremos aquí demostración matemática de esta norma).

Positivo-positivo

```
    00000110   (+ 6)
+   00001000   (+ 8)
---------------------
=   00001110   (+ 14)   O:0     C:0
```

(CORRECTO)

Positivo-positivo con desbordamiento

```
    01111111   (+ 127)
+   00000001   (+ 1)
---------------------
=   10000000   (− 128)   O:1     C:0
```

El resultado es incorrecto, porque se ha producido desbordamiento.

(ERROR)

Positivo-negativo (resultado positivo)

```
    00000100   (+ 4)
+   11111110   (− 2)
---------------------
=(1)00000010   (+ 2)   O:0     C:1 (se desprecia)
```

(CORRECTO)

Positivo-negativo (resultado negativo)

```
    00000010   (+ 2)
+   11111100   (− 4)
---------------------
=   11111110   (− 2)   O:0     C:0
```

(CORRECTO)

Negativo-negativo

```
    11111110   (− 2)
+   11111110   (− 4)
---------------------
= (1)11111010  (− 6)   O:0     C:1 (se desprecia)
```

(CORRECTO)

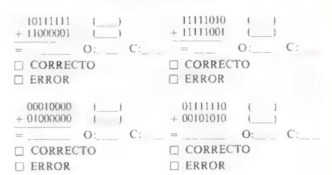
Negativo-negativo con desbordamiento

```
     10000001     (− 127)
+    11000010     (− 62)
= (1)01000011       (67)   O:1      C:1
```

(ERROR)

En este caso se ha producido desbordamiento por adición de dos números negativos muy grandes. El resultado es − 189 y, por su magnitud, no cabe en ocho bits.

Ejercicio 1.12: *Resuélvanse las adiciones propuestas a continuación, indicando en cada caso el resultado, el acarreo C, el desbordamiento O y si el primero es o no correcto.*



```
  10111111    (____)           11111010    (____)
+ 11000001    (____)         + 11111001    (____)
= ______   O:___  C:___      = ______   O:___  C:___
```

☐ CORRECTO ☐ ERROR   ☐ CORRECTO ☐ ERROR

```
  00010000    (____)           01111110    (____)
+ 01000000    (____)         + 00101010    (____)
= ______   O:___  C:___      = ______   O:___  C:___
```

☐ CORRECTO ☐ ERROR   ☐ CORRECTO ☐ ERROR

Ejercicio 1.13: *¿Podría proponer un ejemplo de adición de un número positivo a otro negativo que causase un desbordamiento? ¿Por qué?*

Representación en formato fijo

Ya sabemos representar enteros con signo, pero todavía no hemos resuelto el problema de la magnitud. Para representar enteros grandes necesitamos varios bytes; pero para hacer operaciones artiméticas con eficacia hay que emplear un número de bytes fijo, no variable; por tanto, la determinación del número de bytes supone la del mayor número representable.

Ejercicio 1.14: *¿Cuáles son los números máximo y mínimo representables con dos bytes en complemento a dos?*

El problema de la magnitud

Al sumar números nos hemos limitado a trabajar con ocho bits, porque el microprocesador que vamos a utilizar opera internamente con grupos de ocho bits. Pero esto limita el campo de actuación a las cantidades comprendidas entre − 128 y + 127, sin duda insuficientes para muchas aplicaciones.

Para aumentar el número de cifras representables, se recurre a la precisión múltiple, que consiste en el empleo de formatos de dos, tres o *n* bytes. Veamos algunos ejemplos de un formato de doble precisión de 16 bits:

00000000	00000000	es "0"
00000000	00000001	es "1"
...		
01111111	11111111	es "32767"
11111111	11111111	es "− 1"
11111111	11111110	es "− 2"

Ejercicio 1.15: *¿Cuál es el mayor entero negativo representable con un formato de triple precisión en complemento a dos?*

Pero el método tiene sus inconvenientes. Al sumar dos números, por ejemplo, habitualmente hay que hacerlo de ocho en ocho bits —la forma de operar se describirá en el capítulo 3, Técnicas elementales de programación—, lo que reduce la velocidad del proceso. Además, esta forma de representación adjudica 16 bits a todos los números, incluso a los que cabrían en ocho; en consecuencia, es normal utilizar 16 bits e incluso 32, pero raramente más.

Hay, además, otro punto importante que merece reflexión: sea cual sea el número *n* de bits elegido para la representación en complemento a dos, es fijo. Si el resultado, o un cálculo intermedio, genera un número cuya representación exige más de *n* bits, los que excedan se perderán; por lo general, el programa conservará los *n* de la izquierda (los más significativos) y rechazará los de orden inferior. Esta operación se llama truncar el resultado.

Consideremos el siguiente ejemplo con representación de seis cifras en el sistema decimal:

```
   123456
 ×    1.2
 --------
   246912
  123456
 --------
= 148147.2
```

El resultado necesita de siete cifras, y en el formato en que se trabaja el "2" situado tras el punto decimal se perdería, de manera que el resultado quedaría en 148 147. Por lo general, en la medida en que no se pierde la posición de la coma decimal, este método se utiliza para ampliar la cantidad de operaciones realizables a costa de la precisión.

En el sistema binario el problema es el mismo. Los detalles de la multiplicación binaria se expondrán en el capítulo 4.

La representación en formato fijo, aun con el riesgo de pérdida de precisión que ocasiona, puede bastar para realizar operaciones matemáticas normales.

Por desgracia, la contabilidad no tolera ninguna inexactitud. Cuando se marca el total en una caja registradora, lo que debe aparecer es el precio exacto, no un valor aproximado; por tanto, cuando la precisión es incuestionable, hay que recurrir a otro tipo de representación, que, por lo general, es la llamada BCD, decimal codificado en binario.

Representación BCD

Esta técnica consiste en codificar cada una de las cifras decimales y utilizar todos los bits que sean necesarios para representar con exactitud el número completo. Para codificar las diez cifras comprendidas entre 0 y 9 hacen falta cuatro bits. Tres dan lugar a sólo ocho combinaciones, insuficientes para codificar diez cifras; con cuatro pueden hacerse dieciséis combinaciones, que sí bastan para codificar las diez cifras decimales. Lo malo es que sobran seis posibles códigos (véase figura 1.4), que pueden causar problemas al sumar y restar.

Código	*Símbolo*	*Código*	*Símbolo*
0000	0	1000	8
0001	1	1001	9
0010	2	1010	no utilizado
0011	3	1011	no utilizado
0100	4	1100	no utilizado
0101	5	1101	no utilizado
0110	6	1110	no utilizado
0111	7	1111	no utilizado

Figura 1.4
Tabla BCD.

Como sólo hacen falta cuatro bits para codificar una cifra en BCD, pueden representarse dos cifras en cada byte, formato que se llama *BCD condensado*.

en el formato en que
ecimal se perdería, de
.47. Por lo general, en
n de la coma decimal,
ntidad de operaciones

el mismo. Los detalles
n en el capítulo 4.
aun con el riesgo de
e bastar para realizar

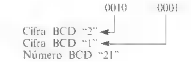
a ninguna inexactitud,
istradora, lo que debe
lor aproximado; por
ble, hay que recurrir a
general, es la llamada

da una de las cifras
sean necesarios para
mpleto. Para codificar
hacen falta cuatro bits.
nes, insuficientes para
acerse dieciséis combi-
s diez cifras decimales.
igos (véase figura 1.4),
y restar.

go	*Símbolo*
0	8
1	9
0	no utilizado
1	no utilizado
0	no utilizado
1	no utilizado
0	no utilizado
1	no utilizado

ara codificar una cifra
en cada byte, formato

Así, "00000000" es "00" en BCD y "10011001" es "99". Un código BCD se lee como sigue:

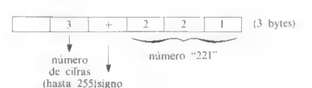


Ejercicio 1.16: *¿Cómo se representa "29" en BCD? ¿Cómo "91"?*

Ejercicio 1.17: *¿Es "10100000" una representación correcta en BCD? ¿Por qué?*

Para representar todas las cifras, se utilizan tantos bytes como sean necesarios. Por lo general, al principio de la representación se reservan uno o más *nibbles* para indicar el número total de *nibbles*, y, por tanto, de cifras BCD. También suele reservarse un *nibble* o un byte para indicar la posición de la coma decimal, aunque las convenciones adoptadas son variables.

He aquí un ejemplo de representación multibyte BCD de un entero:



El número representado es + 221 (en cuanto al signo, 0000 puede representar +, por ejemplo, y 0001, −).

Ejercicio 1.18: *Utilizando la misma convención en cuanto al signo, represéntese el número "− 23 123" en formato BCD, como en el ejemplo, y en notación binaria.*

Ejercicio 1.19: *Represéntese en BCD "222", "111" y el resultado de la operación 222 × 111. (Ejecútese la operación a mano.)*

El sistema BCD se adapta muy fácilmente a la representación de números decimales.

Así, + 2.21 podría representarse como sigue:

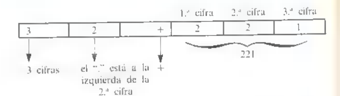


Figura 1.5
Representación en punto [...]tante típica.

La ventaja del sistema BCD es que arroja resultados rigurosamente exactos, aunque a costa de ocupar mucha memoria y reducir la velocidad de ejecución de las operaciones aritméticas. Este inconveniente sólo se acepta en el campo de la contabilidad, y habitualmente el sistema no se emplea en otros casos.

Ejercicio 1.20: *¿Cuántos bits hacen falta para codificar "9999" en BCD? ¿Y en complemento a dos?*

Ya hemos resuelto los problemas asociados a la representación de enteros, de enteros con signo y hasta de grandes enteros. También hemos visto un procedimiento para representar decimales en BCD. Veamos a continuación el problema que supone la representación de números decimales en un formato de longitud fija.

Representación en punto flotante

La condición básica es que los decimales deben representarse en un formato fijo. Para no desperdiciar bits, la representación utilizada pasará por la *normalización* de todos los números.

Al escribir "0.000123" se desperdician tres ceros a la izquierda del número, ceros que sólo sirven para indicar la posición del punto. El número podría normalizarse escribiéndolo .123 $\times 10^{-3}$; ".123" se llama *mantisa normalizada*, y "-3", *exponente*. La normalización ha consistido en la eliminación de todos los ceros situados a la izquierda y en el cálculo del exponente.

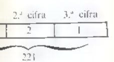
Veamos otro ejemplo: 22.1 se normaliza como $.221 \times 10^2$, y, en general, cualquier número será igual a $M \times 10^E$, siendo M la mantisa y E el exponente.

Como se comprueba inmediatamente, la mantisa de un número normalizado es menor que 1 y mayor o igual que 0.1, siempre que aquél no sea nulo. Es decir:

$$.1 \leqslant M < 1 \quad \text{ó} \quad 10^{-1} \leqslant M < 10^0$$

De la misma manera, en representación binaria:

$$2^{-1} \leqslant M < 2^0 \text{ (ó } .5 \leqslant M < 1)$$



siendo M el valor absoluto de la mantisa con independencia del signo).

Por ejemplo:

$$111.01 \text{ se normaliza como } .11101 \times 2^3$$

siendo la mantisa 11101 y 3 el exponente.

Ahora que ya hemos expuesto el fundamento de la representación, pasemos a examinar el formato real, que aparece en la figura 1.5.

31	24	23	16	15	8	7	0
S	EXP	S	M	A	N	T	I S A

Figura 1.5
Representación en punto flotante típica.

En la representación utilizada en este ejemplo se emplean un total de cuatro bytes con 32 bits. El primer byte por la izquierda contiene el exponente, que, al igual que la mantisa, se representa en complemento a dos: esto significa que el máximo exponente posible es − 128. En la figura 1.5, "S" denota el bit de signo.

Para representar la mantisa se utilizan tres bytes; dado que el primer bit en complemento a dos corresponde al signo, el formato deja 23 bits para albergar la magnitud de la mantisa.

Ejercicio 1.21: *¿Cuántas cifras decimales pueden representarse con una mantisa de 23 bits?*

Esto no es más que un ejemplo de representación en punto flotante. En la práctica pueden usarse sólo tres bytes o más de cuatro. La de cuatro bytes propuesta es una bastante usual, que constituye un compromiso razonable entre exactitud, magnitud representable, aprovechamiento de la memoria y eficacia operativa.

Ya hemos explorado los problemas asociados a la representación de números enteros y decimales, con y sin signo. Podemos, pues, pasar a ocuparnos de la representación interna de datos alfanuméricos.

Representación de datos alfanuméricos

La representación de datos alfanuméricos, es decir, de caracteres, es sencillísima: los caracteres se traducen a un código de ocho bits. En informática no hay más que dos códigos de uso general, el ASCII y el EBCDIC. ASCII son las siglas en inglés de Código Normalizado Americano para el Intercambio de

Información, y es de uso universal en microprocesadores. El código EBCDIC es una variante del ASCII utilizada por IBM que, por tanto, sólo la utilizan los microordenadores en las conexiones con terminales IBM.

Examinemos brevemente el código ASCII. Se trata de codificar 26 letras del alfabeto en minúsculas y mayúsculas, 10 símbolos numéricos y alrededor de 20 símbolos especiales (a efectos de codificación, la Ñ se considera un signo especial, no usado en inglés). Todo ello puede hacerse fácilmente con 7 bits, que proporcionan 128 combinaciones diferentes (véase figura 1.6). Sin embargo, antes hemos hablado de 8 bits. ¿Para qué sirve el octavo? Este octavo bit es el *bit de paridad*, y sirve para garantizar la conservación del contenido de un byte. Funciona de la siguiente manera: se cuenta el total de unos de los siete primeros bits y, si es impar, el octavo se iguala a 1, para que pase [illegible] ser par. Es lo que se llama paridad par (también puede trabajarse con paridad impar, que consiste en calcular el octavo bit —el de la izquierda— de manera que el total de unos sea impar.

Ejemplo: Calcúlese el bit de paridad de "0010011" en paridad par. El número de unos es tres, de manera que el bit de paridad ha de ser 1 para que pase a ser cuatro; por tanto, par: 10010011; el primer 1 es el bit de paridad, y 0010011 es el carácter.

La tabla de códigos ASCII de 7 bits aparece en la figura 1.6. En la práctica se utiliza "tal cual", es decir, sin paridad, añadiendo un 0 en la posición izquierda, o con paridad, añadiendo en esa misma posición el bit adecuado.

HEX CmS	CMS BITS	0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
[illegible]	0000	NUL	DLE	ESPACIO	0	@	P	-	p
1	0001	SOH	DC1	!	1	A	Q	a	q
[illegible]	0010	STX	DC2	"	[illegible]	B	R	b	r
[illegible]	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	$	4	[illegible]	T	d	t
5	0101	ENQ	NAK	%	5	[illegible]	U	[illegible]	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	'	7	G	W	[illegible]	w
[illegible]	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	[illegible]
B	1011	VT	ESC	+	;	K	[	k	{
C	1100	FF	FS	,	<	L	\	l	_
D	1101	CR	GS	—	=	M	]	m	}
E	1110	SO	RS	.	>	N	∧	n	~
F	1111	SI	US	/	?	O	←	o	DEL

Figura 1.6
Tabla de conversión ASCII (véanse las abreviaturas en el apéndice B).

Ejercicio 1.22: *Calcúlese la representación en 8 bits de las cifras "0" a "9" con paridad par (el código resultante se utiliza en los ejemplos de aplicaciones del capítulo 8).*

Ejercicio 1.23: *Idem de las letras "A" a "F".*

Ejercicio 1.24: *Indíquense los contenidos binarios de los cuatro caracteres propuestos a continuación, utilizando para ello el código ASCII sin paridad (el bit de la izquierda es, por tanto, "0"):*



"*A*"
"?"
"*3*"
"*b*"

En situaciones especiales —las telecomunicaciones, por ejemplo— se emplean códigos especiales de corrección y de otras clases, pero están fuera del alcance de este libro.

Una vez estudiadas las formas de representación más usuales en el interior del ordenador para el programa y para los datos, pasaremos a examinar las posibles representaciones externas.

REPRESENTACION EXTERNA DE LA INFORMACION

La representación externa es la forma en que la información se ofrece al *usuario*, que, por lo general, es el programador. Puede ser binaria, octal o hexadecimal y simbólica.

1. Binaria

Como hemos visto, la información se almacena internamente en *bytes*, que son secuencias de ocho *bits* (ceros o unos). A veces es deseable presentar esta información interna directamente en su formato binario; es lo que se llama *representación binaria*. Un ejemplo sencillo lo proporcionan los diodos luminosos, LED, del teclado de algunos microordenadores. Si el microprocesador es de ocho bits, en el teclado habrá probablemente LED para exteriorizar el contenido de cualquiera de los registros (un registro, que se describirá en el capítulo 2, contiene ocho bits de información): un piloto iluminado denota un 1, y otro apagado, un 0. La representación binaria es útil para corregir con detalle programas complejos, sobre todo si incluyen operaciones de entrada y salida, pero constituye una forma de comunicación obviamente poco práctica. Por eso se prefiere

casi siempre la representación simbólica (en efecto, es mucho más fácil entender y recordar "9" que "1001"). Se han ideado formas de comunicación más cómodas que mejoran la relación hombre-máquina.

2. Octal y hexadecimal

En los sistemas octal y hexadecimal se codifican tres y cuatro bits binarios, respectivamente, en un único símbolo. En el primero de ellos, cualquier combinación de tres bits binarios se representa mediante un número comprendido entre 0 y 7: la figura 1.7 recoge la tabla de símbolos de este sistema.

Binario	*Octal*
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Figura 1.7
Símbolos octales.

Así, el número binario "00 100 100" se representa en octal como "044".

0 4 4

Otro caso: 11 111 111 es "377" en octal.

3 7 7

E inversamente, el número octal "211" equivale a

010 001 001

es decir, "10001001" en binario.

El sistema octal se utilizaba tradicionalmente en los ordenadores antiguos, que trabajaban con un número de bits comprendido habitualmente entre 8 y 64. Actualmente, cuando el dominio de los microprocesadores de 8 bits ha convertido a este formato en la norma, resulta más práctico recurrir a la representación *hexadecimal*.

En el sistema hexadecimal, cada grupo de cuatro bits se codifica como una cifra hexadecimal. Estas se representan mediante los símbolos 0 a 9 más las letras A, B, C, D, E y F. Así, "0000" es "0", "0001" es "1" y "1111" es "F" (véase la figura 1.8).

Figura 1.8
Códigos hexadecimales.

a (en efecto. es mucho
"1001"). Se han ideado
que mejoran la relación

nal se codifican tres y
n un único símbolo. En
ión de tres bits binarios
prendido entre 0 y 7; la
de este sistema.

)" se representa en octal

n octal.

211" equivale a

almente en los ordenado-
mero de bits comprendi-
nente, cuando el dominio
convertido a este formato
urrir a la representación

grupo de cuatro bits se
l. Estas se representan
etras A, B, C, D, E y F.
es "F" (véase la figura 1.8).

Decimal	*Binario*	*Hexadecimal*	*Octal*
0	0000	[illegible]	[illegible]
1	0001	1	1
2	0010	[illegible]	2
3	0011	[illegible]	3
4	0100	4	4
5	0101	[illegible]	5
6	0110	6	6
7	0111	7	7
8	1000	[illegible]	10
[illegible]	1001	9	11
10	1010	A	12
11	1011	B	13
12	1100	C	14
13	1101	[illegible]	15
14	1110	[illegible]	16
15	1111	F	17

Figura 1.8
Códigos hexadecimales.

Ejemplo: el número binario 1010 0001 equivale
al hexadecimal A 1

Ejercicio 1.25: *¿Cuál es la representación hexadecimal de "10101010"?*

Ejercicio 1.26: *¿Cuál es el equivalente binario del hexadecimal "FA"?*

Ejercicio 1.27: *¿Cuál es la representación octal de "01000001"?*

El sistema hexadecimal tiene la ventaja de que codifica ocho bits en sólo dos cifras, lo que es más fácil de visualizar y memorizar y más rápido de teclear en el ordenador que el equivalente binario. Por ello, en la mayor parte de los nuevos microordenadores se prefieren representar los grupos de bits en el sistema hexadecimal.

Naturalmente, cuando la información presente en la memoria tenga significado —un texto o una serie de números— el sistema hexadecimal no constituirá un método de representación adecuado para el hombre.

Representación simbólica

Se llama así a la representación externa de la información en su forma simbólica real. Los números decimales, por ejemplo, se representan como tales, y no como secuencias de símbolos hexadecimales o de bits; de la misma manera, el texto escrito adopta la forma de una sucesión de letras. Para el usuario, esta forma de representación es obviamente la más práctica, y se utiliza siempre que se cuente con un dispositivo de visualización adecuado, como un monitor de televisión o una impresora. Por desgracia, tales dispositivos siguen resultando excesivamente costosos para los microordenadores más elementales, que, por tanto, se comunican con el usuario en el sistema hexadecimal.

RESUMEN DE REPRESENTACIONES EXTERNAS

La representación simbólica es la más deseable, ya que es la más natural para el hombre. Sin embargo, exige un interfaz costoso en forma de teclado alfanumérico e impresora o monitor TV, lo que la hace incompatible con los sistemas más baratos. En esas situaciones, la alternativa más frecuente es el sistema hexadecimal. La representación binaria se emplea únicamente en la puesta a punto muy detallada de los soportes lógico o físico; en esta forma de representación se visualiza directamente el contenido de los registros internos de la memoria. (La utilidad de la visualización binaria directa en el panel de mando del ordenador ha sido siempre motivo de acaloradas discusiones, en las que no entraremos aquí.)

Estudiadas las diversas técnicas de representación interna y externa de la información, estamos en condiciones de pasar a examinar el microprocesador que se encargará de manipularla.

EJERCICIOS ADICIONALES

Ejercicio 1.28: *¿Qué ventajas tiene la representación en complemento a dos sobre otras en el manejo de números con signo?*

Ejercicio 1.29: *¿Cómo se representaría "1024" en las notaciones binaria directa, binaria con signo y complemento a dos?*

Ejercicio 1.30: *¿A qué se llama bit O? ¿Debe el programador comprobarlo tras una adición o una sustracción?*

Ejercicio 1.31: *Calcúlense los complementos a dos de "+ 16", "− 17", "+ 18", "− 16", "− 17" y "− 18".*

Ejercicio 1.32: *Indíquese la representación hexadecimal del texto "MENSAJE", almacenado internamente en formato ASCII sin paridad.*



Org

Introducción

2

Organización del hardware del Z80

Introducción

Para hacer programas sencillos no es necesario conocer en detalle la estructura interna del procesador que se está utilizando, pero sí es imprescindible tal conocimiento si se pretenden crear programas más ambiciosos. La finalidad de este capítulo es presentar los aspectos básicos de la constitución física del sistema Z80 indispensables para entender su funcionamiento. Un microordenador completo consta de varios dispositivos, además del microprocesador (el Z80 en este caso); nos limitaremos en esta parte del libro a examinar el Z80 propiamente dicho, y dejaremos el resto de los dispositivos (en su mayor parte componentes de entrada y salida) para más adelante (capítulo 7).

Repasaremos, primero, la arquitectura básica del microordenador; a continuación examinaremos con más detalle la organización interna del Z80 y, en particular, sus diversos registros. A ello seguirá el estudio de la ejecución de programas y del mecanismo de secuenciación. Pero este capítulo dará una visión un tanto simplificada del *hardware;* el lector verdaderamente interesado en ello deberá consultar el libro *Microprocessors,* del mismo autor.

El Z80 se diseñó para sustituir al Intel 8080 y ampliar, de paso, sus posibilidades; a lo largo del texto nos referiremos en varias ocasiones al 8080 de Intel.

Arquitectura del sistema

La arquitectura del microordenador se ilustra en la figura 2.1. El microprocesador (μP), el Z80 en este caso, está situado en la parte izquierda del esquema, y desempeña las funciones de una *unidad central de proceso* (CPU), contenida íntegramente en un solo microcircuito monolítico de silicio; consta de una *unidad aritmética y lógica* (ALU), con sus propios registros internos, y una *unidad de control* (UC), encargada de la secuenciación de las operaciones que ha de realizar el sistema. Veremos su funcionamiento dentro de este mismo capítulo.

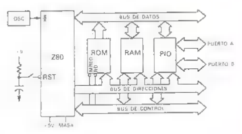


Figura 2.1
Sistema Z80 normal.

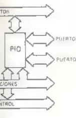
El μP dispone de tres *buses* o canales conductores de información: un *bus de datos* bidireccional de 8 bits, representado en la parte superior de la ilustración; un *bus de direcciones* unidireccional de 16 bits, y un *bus de control*, esquematizado junto con el anterior en la parte inferior de la figura. Veamos ahora la función que cumple cada uno de ellos.

El *bus de datos* transporta datos de unos elementos del sistema a otros, típicamente de la memoria al μP y viceversa, o del μP a circuitos de entrada y salida (estos circuitos de entrada y salida son los componentes encargados de poner el sistema en comunicación con dispositivos externos).

El *bus de direcciones* lleva las direcciones generadas por el μP, que seleccionan entre los registros internos dentro de los microcircuitos conectados al sistema. Dicha dirección especifica la fuente o el destino del dato que debe circular a lo largo del bus de datos.

ntel 8080 y ampliar. de
exto nos referiremos en

or se ilustra en la fi-
Z80 en este caso. está
ema, y desempeña las
ceso (CPU), contenida
monolítico de silicio:
(ALU), con sus propios
ol (UC), encargada de la
a de realizar el sistema.
este mismo capítulo.



canales conductores de
oal de 8 bits, representa-
n; un *bus de direcciones*
control, esquematizado
or de la figura. Veamos
de ellos.
de unos elementos del
oria al μP y viceversa, o
os circuitos de entrada y
. de poner el sistema en
s).
ciones generadas por el
internos dentro de los
icha dirección específica
debe circular a lo largo

El *bus de control* conduce las diversas señales de sincronización que gobiernan el funcionamiento del sistema.

Una vez descrito el papel que desempeñan los *buses*, pasaremos a conectar al sistema los demás componentes necesarios para su funcionamiento.

El μP necesita una referencia de tiempo exacta, que proporcionan un *reloj* y un *cristal*. En los microprocesadores más "antiguos", el reloj-oscilador es externo y, por lo general, adopta la forma de otro circuito integrado. En los modelos más recientes, el mencionado reloj suele ir incorporado en el propio μP. Sin embargo, el cristal de cuarzo, debido a su tamaño, es siempre exterior al sistema. En la figura 2.1, reloj y cristal se han esquematizado a la izquierda del recuadro correspondiente al μP.

Veamos ahora el resto de los elementos del sistema. Avanzando de izquierda a derecha en la figura encontramos: la *memoria sólo de lectura* o *memoria fija (read-only memory, ROM)*, que contiene el *programa* del sistema. La ventaja de la ROM es que su contenido es permanente; no desaparece ni siquiera cuando se desconecta el ordenador. Por ello se utiliza siempre para almacenar un programa de *control* (cuyo funcionamiento explicaremos más adelante) encargado de garantizar la puesta en marcha del sistema. En aplicaciones de control de procesos, casi todos los programas deben almacenarse en ROM, porque normalmente no sufrirán modificaciones y porque, además, deben protegerse frente a cortes de corriente accidentales.

Por el contrario, los aficionados y los profesionales dedicados a la creación, desarrollo y comprobación de programas almacenan éstos en memoria RAM, para poder modificarlos con facilidad. Más adelante, esos programas pueden transferirse a una memoria ROM o dejarse en RAM, aunque el contenido de esta memoria se volatiliza cuando se interrumpe la corriente.

La memoria *RAM (random-access memory, memoria de acceso aleatorio)* es la memoria de lectura y escritura del sistema. En un sistema de control, la capacidad de RAM suele ser reducida, ya que habitualmente se utiliza sólo para almacenar datos; por el contrario, los sistemas destinados al desarrollo de programas necesitan una memoria RAM de gran capacidad, con espacio suficiente para albergar programas y material lógico en desarrollo. El contenido de la memoria RAM debe cargarse de un dispositivo externo antes de usarla.

Por último, el ordenador necesita uno o más circuitos integrados de conexión para comunicarse con el mundo exterior. El circuito de conexión más usado es el llamado *PIO (parallel input/output, entrada y salida en paralelo)*, que se muestra en la figura. Este PIO, como todos los demás circuitos del sistema, está conectado a los tres *buses* y proporciona, al menos,

dos puertos de 8 bits para facilitar la comunicación con el mundo exterior. Para más detalles sobre el funcionamiento real del PIO, consulte el libro *Microprocessors;* si le interesa su aplicación específica al sistema Z80, más adelante, en el capítulo 7 de este libro (Dispositivos de entrada y salida), se trata.

Todos los microcircuitos están conectados a los tres *buses*, incluido en todos los casos el de control.

Pero no todos los módulos funcionales que acabamos de describir tienen que estar integrados en una misma pastilla LSI *(large scale integration, de integración a gran escala).* Lo normal es, por el contrario, combinar varios circuitos, como un PIO, y varias ROM o RAM.

Acabamos de describir los componentes esenciales, pero para que el sistema funcione hacen falta algunos más: así, los *buses* deben estar, por lo general, *protegidos* por amplificadores-separadores; los microcircuitos RAM necesitan un *circuito lógico de decodificación;* por último, algunas señales deben ser, asimismo, intensificadas por *separadores.* Aquí no describiremos estos circuitos auxiliares, puesto que no influyen en la programación. El lector interesado en las técnicas de montaje y creación de conexiones deberá consultar el volumen, *Microprocessor Interfacing Techniques.*

Figura 2.2
Estructura habitual de un microprocesador.

El interior del microprocesador

La inmensa mayoría de los circuitos integrados microprocesadores que actualmente existen en el mercado tienen la misma arquitectura que describiremos aquí y que muestra la figura 2.2. Empezaremos a examinar con detalle los módulos que componen el microprocesador a partir del extremo derecho de la ilustración.

La casilla de *control* situada en la parte derecha representa la unidad de control, encargada de sincronizar la actividad de todo el sistema. Su comportamiento se irá aclarando a lo largo de lo que queda de capítulo.

La *ALU* realiza operaciones aritméticas y lógicas. Una de las entradas de la ALU, la situada a la izquierda en este caso, está equipada con un registro especial llamado acumulador (puede haber varios acumuladores). Dentro de la misma instrucción, el acumulador puede referenciarse como entrada y como salida (fuente y destino).

Figura 2.3
Desplazamiento y rotación

La ALU se encarga también de las operaciones de *desplazamiento* y *rotación* de bits.

Se llama desplazamiento a la traslación del contenido de un byte una o más posiciones hacia la izquierda o hacia la derecha

Figura 2.2
Estructura habitual de un microprocesador.

(véase la figura 2.3). Cada uno de los bits avanza una posición hacia la izquierda. Los detalles de estas dos instrucciones se estudiarán en el próximo capítulo.

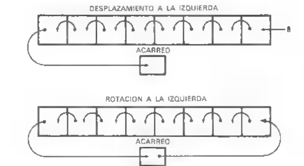


Figura 2.3
Desplazamiento y rotación.

Nota: Algunas instrucciones de desplazamiento y rotación no incluyen acarreo.

El desplazador puede estar situado en la salida de la ALU, como en la figura 2.2, o en la entrada del acumulador.

A la izquierda de la ALU están las *banderas* o *registro de estado*. Su función es "llevar la cuenta" de las situaciones excepcionales que se dan en el interior del microprocesador. El conte-

nido del registro de estado puede verificarse mediante instrucciones especiales o leerse en el bus interno de datos. Las instrucciones *condicionales* provocan la ejecución de un nuevo programa en función del valor de uno de esos bits.

La función de los bits de estado en el Z80 se estudiará más adelante en este mismo capítulo.

ACTIVACION DE LAS BANDERAS

La mayor parte de las instrucciones ejecutadas por el procesador modificarán alguna de las banderas o todas ellas. Es importante consultar siempre la tabla del fabricante, que indica qué bits serán modificados por las instrucciones, ya que ésa es la única forma de comprender cómo se va desarrollando el programa. La figura 4.17 recoge una tabla como la mencionada para el Z80.

LOS REGISTROS

Volvamos a la figura 2.2. A la izquierda del esquema se observan los registros del microprocesador, dividios conceptualmente en dos categorías: *registros de tipo general* y *registros de direcciones*.

REGISTROS DE TIPO GENERAL

Los registros de tipo general deben organizarse en orden para que la ALU manipule los datos a velocidad elevada. Debido a las limitaciones del número de bits que resulta razonable proporcionar dentro de una instrucción, casi siempre hay menos de ocho registros (directamente direccionables). Cada uno de ellos es un conjunto de ocho elementos biestables conectados al *bus* bidireccional interno de datos. Los ocho bits pueden entregarse simultáneamente al mencionado *bus* o recibirse desde el mismo. Los registros biestables en MOS (tipo de dispositivo, *metal-óxido-semiconductor*) constituyen la forma de memoria más rápida existente, y el acceso a su contenido se lleva a cabo en algunas decenas de nanosegundos.

Los registros *internos* están habitualmente etiquetados de 0 a *n*, y su función no está definida de antemano (por eso se dice que son de tipo general). Pueden contener cualquier dato utilizado por el programa.

Figura 2.4
Los registros de direccion
16 bits crean el *bus* de dire
nes.

Estos registros de tipo general suelen utilizarse para almacenar datos de ocho bits. En algunos microprocesadores pueden manipularse *dos* de esos registros simultáneamente, que se llaman "registros apareados" y que permiten el almacenamiento de magnitudes —datos o direcciones— de 16 bits.

REGISTROS DE DIRECCIONES

Son registros de 16 bits destinados específicamente al almacenamiento de direcciones, que con frecuencia se conocen también como *contadores de datos* o *apuntadores*. Su característica principal es que están conectados al *bus* de direcciones. De hecho, los registros de direcciones crean el *bus* de direcciones (éste aparece en la parte inferior izquierda de la figura 2.4).

Estos registros de 16 bits sólo pueden cargarse por medio del *bus* de datos, lo que obliga a realizar dos transferencias de 8 bits. Para diferenciar las mitades inferior y superior de cada registro, suele llamarse a la primera L (inferior, *lower*, que comprende los bits 0 a 7), y H (superior, *higher*, que comprende los bits 8 a 15), a la segunda; tales etiquetas se emplean siempre que es necesario distinguir entre las dos mitades. En la mayor parte de los microprocesadores hay, al menos, dos registros de direcciones (véase la figura 2.4; "MUX" es abreviatura de multiplexor).

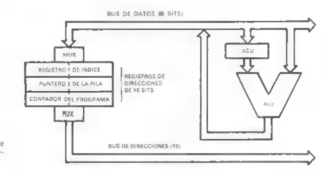


Figura 2.4
Los registros de direcciones de 16 bits crean el *bus* de direcciones.

Contador del programa (PC)

Es un elemento imprescindible en cualquier procesador. Contiene la dirección de la instrucción que ha de ejecutarse a continuación. El mecanismo de ejecución del programa y la

secuenciación automática que se lleva a cabo con ayuda del contador del programa se describirán en la siguiente sección. Por el momento, diremos que la ejecución de un programa es normalmente secuencial: para acceder a la instrucción siguiente es preciso extraerla primero de la memoria interna del microprocesador. Para ello, se entrega el contenido del PC al *bus* de direcciones, que lo transmite a la memoria: ésta lee el contenido especificado por la dirección y devuelve la palabra correspondiente, que es la instrucción, a la µP. En unos pocos microprocesadores, como el F8 de dos microcircuitos integrados, no hay PC. Esto no significa que el sistema no tenga ese elemento, sino que, por razones de eficacia, actúa directamente en el circuito integrado de memoria.

Puntero de la pila SP

La *pila* se describirá en la próxima sección. En los microprocesadores de tipo general más potentes, la pila suele realizarse en la memoria mediante el soporte lógico. Para identificar el elemento superior de la pila dentro de aquélla se reserva un registro de 16 bits llamado *puntero de la pila o SP (stack pointer)*. El SP contiene la dirección del elemento superior de la pila dentro de la memoria. Como se verá, la pila es indispensable para programar interrupciones y acceder a las subrutinas.

Registro de índice (IX)

La adjudicación de índices es una operación de direccionamiento que no está presente en todos los microprocesadores. En el capítulo 5 se describirán las diversas técnicas de direccionamiento. El empleo de índices permite acceder con una sola instrucción a bloques completos de datos contenidos en la memoria. El *registro de índice* suele contener un valor de desplazamiento que se suma automáticamente a una base (o viceversa, una base que se añade a un desplazamiento); de esta forma, el índice da acceso a cualquiera de las palabras de un bloque de datos.

LA PILA

Una *pila* es lo que formalmente se llama una estructura LIFO (*last-in first-out*, último en entrar, primero en salir). Está formada por un conjunto de registros, o posiciones de memoria, adscritos a dicha estructura de datos. Su característica esencial es que se trata de una estructura *cronológica*; el primero de los elementos introducidos en la pila ocupa siempre el fondo de la misma, mientras que la introducción más reciente está siempre en su parte superior. Su organización es comparable a la del

portabandejas de la barra de una hamburguesería; las bandejas se apilan sobre una base que se hunde en el pozo de la barra conforme aumenta el peso: se colocan y se cogen por arriba, de manera que las últimas en llegar al montón son siempre las primeras en salir de él. Este ejemplo ilustra también otra peculiaridad de la pila, a saber: que sólo es accesible mediante dos instrucciones: empujar y extraer (PUSH y POP).

La operación de *empujar* consiste en la carga de un elemento (dos en el caso del Z80) en lo alto de la pila. Por *extracción* se entiende la toma de un elemento de la pila. En un microprocesador es el *acumulador* lo que se coloca en la parte superior de la pila, de manera que la extracción consiste en la transferencia a éste del último elemento de aquélla. Puede haber también otras instrucciones para llevar el elemento superior de la pila a otros registros particulares, como el de estados, por ejemplo; a este respecto, el Z80 [redacted] más flexible que casi todos los demás microprocesadores.

La pila es necesaria para aplicar tres recursos de programación: subrutinas, interrupciones y almacenamiento temporal de datos. El papel que desempeña la pila en la ejecución de subrutinas se explicará en el capítulo 3 (Técnicas básicas de programación). Su función en las interrupciones la veremos en el capítulo 6 (Técnicas de entrada y salida). Por último, el comportamiento de la pila como almacenamiento temporal de datos en el proceso a alta velocidad se expondrá durante los programas de aplicaciones específicas.

Por el momento, nos contentarmeos con aceptar que la pila es un componente imprescindible en cualquier sistema informático. Se crea de dos formas:

1. La primera consiste en reservar un número fijo de registros dentro del propio microprocesador, es lo que se llama una pila en soporte físico. Tiene la ventaja de la velocidad y el inconveniente del número limitado de registros.
2. En casi todos los microprocesadores de tipo general la pila se realiza en el soporte lógico para no limitarla a un número muy reducido de registros. Es el procedimiento empledo en el Z80. Dentro del microprocesador se reserva un registro, el SP en este caso, para almacenar el puntero de la pila, es decir, la dirección de su elemento superior (o, a veces, la dirección del elemento superior más uno). A continuación se crea la pila como una zona de la memoria; de esta forma bastan los 16 bits que ocupa el puntero para señalar cualquier posición de aquélla.

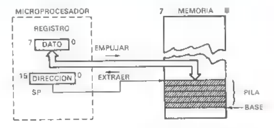


Figura 2.5
Las dos instrucciones de manipulación de la pila.

CICLO DE EJECUCION DE LAS INSTRUCCIONES

Examinemos la figura 2.6. La unidad microprocesadora aparece a la izquierda, y la memoria, a la derecha. Esta puede estar formada por microcircuitos ROM o RAM o de cualquier otra clase con capacidad de almacenamiento de datos e instrucciones. Veremos en este caso cómo se toma una instrucción de la memoria para ilustrar la función del contador del programa; supondremos que el contenido de éste es válido; ahora tiene una dirección de 16 bits, que corresponde a la siguiente instrucción, la que debe tomarse de la memoria. Todos los procesadores siguen un ciclo de tres partes:

1. Tomar la siguiente instrucción.
2. Decodificar la instrucción.
3. Ejecutar la instrucción.

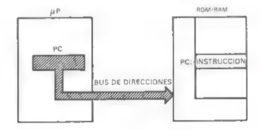
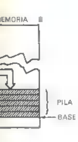


Figura 2.6
Tomar una instrucción de la memoria.

Figura 2.7
Secuenciación automática

Tomar

En la primera parte del ciclo, el contenido del contador del programa se deja en el *bus* de direcciones, que lo conduce a la memoria. Simultáneamente, caso de que sea necesario, se emite una señal de lectura a lo largo del *bus* de control del sistema.



NSTRUCCIONES

microprocesadora apa-
recha. Esta puede estar
M o de cualquier otra
de datos ■ instruccio-
una instrucción de la
ontador del programa;
es válido: ahora tiene
■ a la siguiente instruc-
. Todos los procesado-



enido del contador del
es, que lo conduce a la
sea necesario, se emite
de control del sistema.

La memoria recibe la dirección, que especifica una de sus posiciones. Al recibir la señal de lectura, decodifica la dirección por medio de su propio decodificador y selecciona la posición indicada en ella. Algunos cientos de nanosegundos más tarde, la memoria deja el dato de ocho bits correspondiente a la dirección pedida en el *bus* de datos. Esta palabra de ocho bits es la instrucción que se quería tomar. En la figura, la instrucción se entrega al *bus* de datos esquematizado en la parte superior del μP.

Resumamos brevemente la secuencia: el contenido del contador del programa pasa al *bus* de direcciones; se genera una señal de lectura; *la memoria realiza su ciclo* y, unos trescientos nanosegundos más tarde, la instrucción situada en la dirección pedida pasa al *bus* de datos (suponiendo que la instrucción ocupe un solo byte). El microprocesador lee el *bus* de datos y deposita su contenido en un registro interno especial llamado IR o *registro de instrucción*; es un registro de ocho bits que se utiliza para almacenar la instrucción que se acaba de tomar de la memoria. El ciclo tomar ya se ha cerrado. Los ocho bits de la instrucción ya están físicamente alojados en el registro interno especial, IR, del μP. Este registro, situado a la izquierda en la figura 2.7, no es accesible al programador.

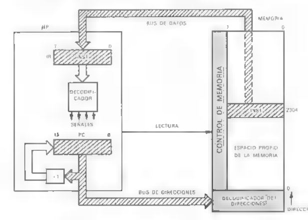


Figura 2.7
Secuenciación automática.

Decodificación y ejecución

Una vez alojada la instrucción en el IR, la unidad de control decodifica su contenido y genera la secuencia de señales internas y externas necesarias para ejecutar la instrucción especificada. Hay, pues, un breve retardo de decodificación al que sigue una

fase de ejecución, cuya duración depende de la naturaleza de la instrucción. Algunas se ejecutan íntegramente dentro del μP, mientras que otras toman datos de la memoria o los dejan en ella. Por eso, cada tipo de instrucción necesita un tiempo de ejecución diferente. Este tiempo se mide en ciclos de reloj (en el capítulo 4 se detallan los ciclos empleados por cada una de las instrucciones). Los tiempos se expresan en ciclos, y no en nanosegundos, porque la duración del ciclo es variable.

TOMAR LA SIGUIENTE INSTRUCCION

Ya hemos descrito cómo, con ayuda del contador del programa, se toma una instrucción de la memoria. Durante la ejecución de un programa, las instrucciones se toman de la memoria *en secuencia*, y para ello es preciso prever un mecanismo automático, que adopta la forma de un sencillo sumador conectado al contador del programa (figura 2.7). Su funcionamiento es el siguiente: cada vez que se coloca en el *bus* de direcciones el contenido del contador del programa (parte inferior de la ilustración), dicho contenido se incrementa en una unidad y se vuelve a escribir en el contador. Si, por ejemplo, el contador tiene el valor "0", éste pasa al *bus* de direcciones, mientras aquél pasa a valer "1". De esta forma, cuando vuelva a funcionar traerá la instrucción de la dirección 1. Este ciclo constituye un *mecanismo automático de secuenciación de instrucciones*.

La descripción del párrafo anterior es, en realidad, una simplificación de la realidad, porque algunas instrucciones pueden tener dos y hasta tres bytes de longitud, lo que obliga a tomarlos de la memoria uno tras otro; sin embargo, el mecanismo fundamental es idéntico: el contador del programa se usa tanto para tomar bytes sucesivos de una instrucción como para tomar sucesivas instrucciones. Por tanto, el dispositivo formado por el contador y el sumador señala automáticamente posiciones sucesivas de la memoria.

Vamos ahora a ejecutar una instrucción dentro del μP (véase la figura 2.8). Una instrucción típica sería, por ejemplo, R0 = R0 + R1, que significa: "SUMAR el contenido de R0 al de R1 y almacenar el resultado en R0". Para realizar la operación, se lee el contenido del registro R0, que, a través del *bus* único, se lleva a la entrada izquierda de la ALU y se almacena en el registro auxiliar* situado en dicho punto. A continuación pasa al *bus* el contenido de R1, que llega a la ALU por su acceso derecho (véase la secuencia en las figuras 2.9 y 2.10). En

* Se ha traducido la expresión *buffer* por memoria o registro auxiliar, pudiéndose encontrar en otros textos como registro tampón o registro intermedio.

Figura 2.8
Arquitectura de bus únic

Figura 2.9
Ejecución de una suma: F
sa a ACU.

Figura 2.10
Suma. ■ segundo regist
pasa a ALU.

de la naturaleza de la
gramente dentro del
le la memoria o los
strucción necesita un
se mide en ciclos de
s empleados por cada
expresan en ciclos, y
del ciclo es variable.

ION

del contador del pro-
memoria. Durante la
ones se toman de la
so prever un mecanis-
un sencillo sumador
ura 2.7). Su funciona-
coloca en el *bus* de
programa (parte infe-
se incrementa en una
or. Si, por ejemplo, el
al *bus* de direcciones.
orma, cuando vuelva a
ión 1. Este ciclo cons-
iación de instrucciones.
es, en realidad, una
nas instrucciones pue-
gitud, lo que obliga a
n embargo, el mecanis-
del programa se usa
instrucción como para
el dispositivo formado
omáticamente posicio-

ucción dentro del μP
ica sería, por ejemplo,
el contenido de R0 al
Para realizar la opera-
que, a través del *bus*
la ALU y se almacena
punto. A continuación
ega a la ALU por su
figuras 2.9 y 2.10). En

emoria o registro auxiliar.
egistro tampón [redacted] registro

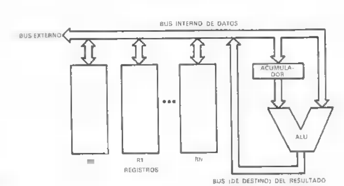


Figura 2.8
Arquitectura de bus único.

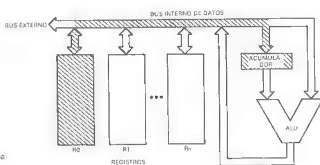


Figura 2.9
Ejecución de una suma. R0 pa-
[redacted] a ACU.

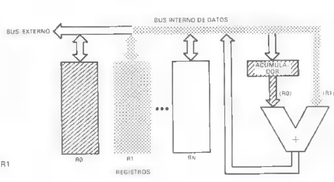


Figura 2.10
Suma: [redacted] segundo registro R1
pasa [redacted] ALU.

este momento, la entrada derecha de la ALU viene determinada por R1, y la izquierda, por el registro auxiliar, que contiene el valor anterior de R0; acto seguido se realiza la suma en la UAL, y el resultado pasa a la salida de la misma (véase la figura 2.11, parte inferior derecha). Allí lo recoge el *bus* único y lo transporta hasta R0; en la práctica, esto significa que la entrada de R0 está abierta y que el dato puede escribirse en su interior. El resultado de la suma ya está en R0, y la ejecución de la instrucción ha terminado. Obsérvese que el contenido de R1 no se ha modificado; esto ilustra un principio general: la operación de lectura no modifica los contenidos de los registros ni de la memoria de lectura/escritura.

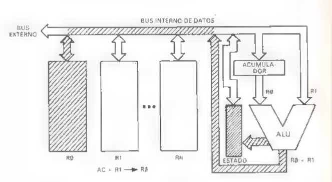


Figura 2.11
El resultado que acaba de generarse va a R0.

El registro auxiliar de la entrada izquierda de la ALU sirve para *memorizar* el contenido de R0, mientras el *bus*, que es único, recuérdese, se emplea para transferir otros contenidos; pero sigue habiendo un problema.

EL PROBLEMA DE LA VELOCIDAD CRITICA

La sencilla organización ilustrada en la figura 2.8 no funcionaría correctamente.

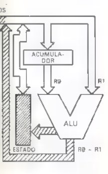
Pregunta: *¿En qué consiste el problema de la sincronización?*

Respuesta: El problema está en que el resultado producido por la ALU pasa al *bus* único y se propaga a todo lo largo del mismo, no únicamente en dirección de R0; en concreto, volverá a determinar la entrada derecha de la ALU y modificará el

Figura 2.12
El problema de la velocidad
tica.

Figura 2.13
Los dos registros auxiliare
cesarios (o registros tem
les).

LU viene determinada
xiliar, que contiene el
realiza la suma en la
le la misma (véase la
recoge el *bus* único y
esto significa que la
puede escribirse en su
en R0, y la ejecución
que el contenido de
principio general: la
enidos de los registros



ierda de la ALU sirve
entras el *bus*, que es
erir otros contenidos;

D CRITICA

a figura 2.8 no funcio-

de la sincronización?

sultado producido por
ga a todo lo largo del
0; en concreto, volverá
ALU y modificará el

resultado, que irá a la salida unos pocos nanosegundos más tarde. En esto consiste el problema de la *velocidad crítica*. Es imprescindible aislar la salida de la ALU de su entrada (véase la figura 2.12).

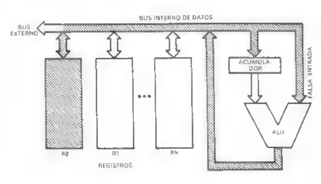


Figura 2.12
El problema de la velocidad crítica.

Para ello hay varias soluciones, siempre utilizando un registro auxiliar. Este puede instalarse tanto a la salida como a la entrada de la ALU; por lo general, se coloca a la entrada, en este caso en su lado derecho. Ahora el sistema funcionará ya correctamente. Más adelante, en este mismo capítulo, veremos que si el registro ilustrado en el lado izquierdo se usa como acumulador (para instrucciones de un byte de longitud), necesitará, a su vez, uno auxiliar, como el representado en la figura 2.13.

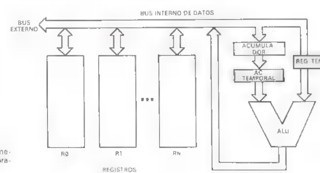


Figura 2.13
Los dos registros auxiliares necesarios (o registros temporales).

Organización interna del Z80

Los términos necesarios para entender los elementos internos del microprocesador están ya definidos, de manera que podemos pasar a examinar con más detalle el Z80 propiamente dicho y a describir sus posibilidades. La organización interna del Z80 aparece en la figura 2.14, que constituye una descripción lógica del dispositivo. Puede haber algunas otras interconexiones adicionales, pero no se han mostrado. Empezaremos a seguir el esquema a partir de la derecha.

Lo primero que encontramos es la *unidad aritmética y lógica* (la ALU), fácilmente reconocible por su característica forma de V. El acumulador, del que ya hablamos en la sección anterior, situado en la entrada derecha de la ALU, se identifica como A. También vimos en esa sección que el acumulador debe contar con un *registro auxiliar*, identificado en la figura como ACT (acumulador temporal). También la entrada izquierda de la ALU dispone de un *registro temporal*, denominado TMP. El funcionamiento de la ALU se aclarará en la próxima sección, en la que describiremos la ejecución de instrucciones reales.

En el Z80, el *registro de estado* se llama F, y aparece a la derecha del acumulador. Su contenido está básicamente determinado por la ALU, pero veremos que algunos de sus bits pueden también depender de otros módulos o de otros sucesos.

Los registros acumulador y de estado aparecen duplicados e identificados respectivamente, A, A' y F, F'; ello se debe a que el Z80 está quipado internamente de dos juegos de registros: A + F y A' + F', aunque sólo puede usarse *un* juego a la vez; hay una instrucción especial para intercambiar los contenidos de A y F con los de A' y F'. Para simplificar las explicaciones, en la mayor parte de los esquemas ulteriores representaremos únicamente A y F; el lector deberá recordar que dispone, si lo desea, de la opción de pasar a los registros A' y F'.

La función de cada una de las banderas del registro F se describirá en el capítulo 3 (Técnicas básicas de programación).

En el centro de la figura aparece un amplio bloque de registros, en cuya parte superior se observan dos grupos idénticos. Cada uno de ellos dispone de seis registros, identificados por las letras B, C, D, E, H y L; se trata de los *registros de tipo general de ocho bits* del Z80. Este tiene dos peculiaridades que lo diferencian del microprocesador general descrito al comienzo de este capítulo.

En primer lugar, el Z80 dispone de *dos* bancos de registros, es decir, de dos grupos idénticos de seis registros cada uno. En un momento determinado, sólo pueden usarse seis registros, pero hay instrucciones especiales para intercambiar sus conteni-

Figura 2.14
Organización interna del

r los elementos inter-
idos, de manera que
le el Z80 propiamente
organización interna
onstituye una descrip-
gunas otras interconc-
rado. Empezaremos a
a.
dad aritmética y lógica
aracterística forma de
en la sección anterior,
, se identifica como A.
umulador debe contar
la figura como ACT
rada izquierda de la
denominado TMP. El
la próxima sección, en
trucciones reales.
ama F, y aparece a la
stá básicamente deter-
e algunos de sus bits
dos o de otros sucesos.
aparecen duplicados e
F'; ello se debe a que
s juegos de registros:
rse *un* juego a la vez:
ambiar los contenidos
ificar las explicaciones,
riores representaremos
rdar que dispone, si lo
stros A' y F'.
leras del registro F se
icas de programación).
mplio bloque de regis-
dos grupos idénticos,
os, identificados por las
registros de tipo general
peculiaridades que lo
descrito al comienzo de

los bancos de registros,
registros cada uno. En
usarse seis registros,
tercambiar sus conteni-

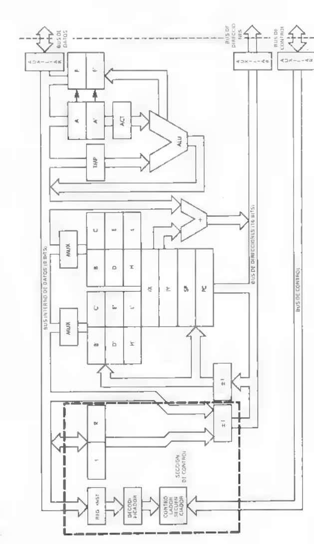


Figura 2.14
Organización interna del Z80.

dos ente los dos grupos. Por tanto, uno de ellos se comporta como memoria interna, mientras el otro funciona como bloque de registros; las posibles aplicaciones de esta peculiaridad se descibirán en el próximo capítulo.

Para evitar confusiones, supondremos, a partir de ahora, que sólo hay seis registros de trabajo —B, C, D, E, H y L— e ignoraremos los del segundo grupo.

El símbolo MUX, que aparece en la parte superior de la memoria, es abreviatura de *multiplexor;* los datos procedentes del *bus* interno de datos pasan al registro seleccionado a través de ese elemento. En un momento determinado, sólo puede haber un registro conectado al *bus* interno.

La segunda peculiaridad de estos seis registros, además de ser registros generales de ocho bits, es que disponen de una conexión con el *"bus" de direcciones*, y por eso se agrupan por *pares*. En efecto, los contenidos de B y C, por ejemplo, pueden entregarse simultáneamente al *bus* de direcciones de 16 bits, ilustrado en la parte inferior de la figura. Gracias a ello, los seis registros de este grupo pueden usarse tanto para almacenar datos de ocho bits como para guardar apuntadores de dieciséis bits para el direccionamiento de la memoria.

El tercer grupo de registros, que en la ilustración aparece en el centro, bajo los dos anteriores, contiene cuatro registros de direcciones "puros". Como en cualquier microprocesador, hay un contador del programa (PC) y un puntero de la pila (SP). Recuérdese que en el contador del programa está la dirección de la instrucción que ha de ejecutarse acto seguido.

El puntero de la pila señala la parte superior de la pila de la memoria. En el caso del Z80, señala la *última entrada real* en la pila (en otros microprocesadores señala justo por encima de la última entrada). La pila crece *hacia abajo*, es decir, hacia las direcciones más bajas.

Esto significa que el puntero debe *disminuir* cada vez que se *introduce* una nueva palabra en la pila. Y viceversa, cuando se *extrae* una palabra de la pila, el valor del puntero *aumenta* en uno. En el Z80, las operaciones de introducir y extraer siempre implican *dos* palabras al mismo tiempo, por lo que el valor del puntero disminuye y aumenta en esa medida.

Faltan por describir dos registros del grupo central de cuatro: se trata de dos *registros índices* llamados IX (registro índice X) e IY (registro índice Y), equipados ambos con un sumador especial representado como una ALU pequeña en forma de V, a la derecha de los mismos (seguimos en la figura 2.14). Cuando se usa una instrucción indexada, por el *bus* de datos interno se mueve un byte especial llamado *desplazamiento*, que se suma a los contenidos de IX o IY. Hay instrucciones

especiales que automáticamente suman este desplazamiento a IX o IY y generan una dirección; es lo que se llama *indexación*, y permite acceder cómodamente a cualquier bloque de datos secuenciales. Esta valiosa opción se describirá en el capítulo 5 (Técnicas de direccionamiento).

En la parte inferior del bloque central de registros, y a su izquierda, hay una casilla señalada "± 1"; se trata de un sumador/restador que incrementa o decrementa automáticamente el contenido de cualquiera de los pares de registros SP, PC, BC, DE, o HL (los registros de direcciones "puros") cada vez que pasan una dirección al *bus* interno de direcciones. Es un dispositivo indispensable para realizar los *bucles de programa* automatizados que describiremos en la próxima sección. Gracias [redacted] él se puede acceder cómodamente a posiciones sucesivas de la memoria.

Pasemos ahora al extremo izquierdo de la figura. Vemos un par de registros aislados identificados como I y R. El I se llama *registro de interrupción-dirección de página*, y su función se describirá en la sección dedicada [redacted] las interrupciones del capítulo 6 (Técnicas de entrada y salida); se utiliza solamente en un caso especial en el que, como respuesta a una interrupción, se genera una llamada indirecta a una posición de memoria. El registro I que nos ocupa almacena la parte superior de la dirección indirecta; la parte inferior de ésta la proporciona el dispositivo que provoca la interrupción.

El registro R es el de *refresco de la memoria*, y sirve para refrescar automáticamente las memorias dinámicas. Como está asociado a la memoria dinámica, este registro suele montarse fuera del microprocesador; es un elemento muy cómodo, que reduce el soporte físico necesario para algunos tipos de memorias dinámicas. No la utilizaremos aquí con fines de programación, puesto que se trata fundamentalmente de una característica del soporte físico (véase *Microprocessor Interfacing Techniques*, donde [redacted] da una descripción detallada de las técnicas de refresco de la memoria); no obstante, puede utilizarse también como reloj del soporte lógico, por ejemplo.

Pasemos ahora al extremo izquierdo de la ilustración, donde aparece la sección de control del microprocesador. Empezando por arriba, primero encontramos el *registro de instrucción*, IR, que contiene la instrucción que ha de ejecutarse (este registro no tiene nada que ver con el par "I, R" descrito más arriba). La instrucción se recibe de la memoria por medio del *bus* de datos, se transmite a lo largo del *bus* de datos interno y, por fin, se deposita en el registro de instrucción. Bajo éste se encuentra el *decodificador*, que envía señales al controlador-secuenciador y determina la ejecución de la instrucción dentro y fuera del

microprocesador. La *sección de control* genera y gobierna el *bus* de control ilustrado en la parte inferior de la figura.

Los tres *buses* gobernados por el sistema —el de datos, el de direcciones y el de control— se prolongan fuera del microprocesador por medio de las patillas de conexión del mismo: estas conexiones aparecen en el extremo derecho de la figura, y, como se ve, están aisladas del exterior por separadores.

Ya hemos descrito todos los elementos lógicos del Z80. Para empezar a escribir programas no es necesario conocer al detalle el funcionamiento del microprocesador, pero quien esté interesado por escribir códigos eficaces deberá entender de qué forma se ejecutan las instrucciones en su interior, porque la velocidad y el tamaño del programa dependen de la elección acertada de registros y técnicas; por tanto, examinaremos a continuación la ejecución de algunas instrucciones típicas en el interior del Z80, y veremos así el funcionamiento y la utilización de los registros internos y los *buses*.

Formatos de las instrucciones

Figura 2.15
Formatos típicos de instru-
nes.

Las instrucciones del Z80 aparecen en el capítulo 4 y pueden tener uno, dos, tres o cuatro bytes. Una instrucción especifica la operación que debe llevar a cabo el microprocesador. Desde un punto de vista simple, cualquier instrucción puede representarse mediante un código de operación seguido por un campo opcional literal o de dirección, que consta de una o dos palabras. El código del campo de operación especifica la que ha de emprenderse. En términos informáticos estrictos, el código de operación consta únicamente de los bits que especifican la operación que debe realizarse, con exclusión de los punteros que pudieran ser necesarios. Pero en el ámbito de los microprocesadores es útil llamar código de operación al conjunto de éste más los punteros. Este "código de operación generalizado" debe residir en una palabra de ocho bits para alcanzar la máxima eficacia (ya que éste es el factor limitante del número de instrucciones de que dispone el microprocesador).

El 8080 utiliza instrucciones de uno, dos o tres bytes de longitud (véase la figura 2.15). Pero el Z80, que dispone de instrucciones adicionales indexadas, necesita un byte más. Los códigos de operación del Z80 son, por lo general, de un byte de longitud, salvo en el caso de instrucciones especiales, de dos bytes.

En algunas instrucciones, al código de operación debe seguir un byte de datos, de manera que la instrucción pasará a ser de

dos bytes (salvo que haya indexación, lo que supone otro byte más).

Hay ocasiones en que la instrucción exige la especificación de una dirección. Como éstas ocupan 16 bits —dos bytes—, la instrucción tendrá tres o cuatro bytes.

Para cada byte de instrucción, la unidad de control tendrá que ejecutar una operación de tomar de la memoria, que tarda en cumplirse cuatro ciclos de reloj. Como es obvio, cuanto más corta sea la instrucción, tanto más rápida será la ejecución.



Figura 2.15
Formatos típicos de instrucciones.

INSTRUCCIONES DE UNA PALABRA

Estas instrucciones son, en principio, las más rápidas y las preferidas por los programadores. Una instrucción monopalabra típica del Z80 sería:

LD r, r′

que significa: "transferir el contenido del registro r′ a r". Se trata de una operación registro a registro. Todos los microprocesadores necesitan esta clase de instrucciones, que permiten al programador transferir información de cualquiera a cualquiera de los registros de la máquina. Las que se refieren a registros especiales, como el acumulador u otros, pueden necesitar de un código de operación peculiar.

Una vez ejecutada la instrucción anterior, el contenido de r será igual que el de r′ que, por su parte, *no* se habrá visto modificado durante la operación de lectura.

Todas las instrucciones se representan internamente en formato binario, y la notación "LD r, r′" es simbólica o *mnemónica* y se llama representación de una instrucción en *lenguaje ensamblador*. Se trata, simplemente, de una codificación simbólica más cómoda de utilizar que el verdadero código binario, que en el caso que nos ocupa sería: 01DDDFFF (bits 0 a 7).

Esta representación sigue siendo parcialmente simbólica. Cada una de las letras D y F representa un bit binario; las tres D, "DDD", corresponden a los tres bits que señalan el registro de *destino*. Bastan tres bits para seleccionar uno de los ocho registros posibles; los códigos de dichos registros aparecen en la figura 2.16; el correspondiente al registro B, por ejemplo, es "000", "001" el de C, etc.

De la misma manera "FFF" son los tres bits que señalan el registro fuente. En este caso, la convención es que el registro r' sea la fuente y el r, el destino. Los bits de la representación binaria de una instrucción no se organizan a gusto del programador, sino en función de las exigencias de la sección de control del microprocesador, que debe decodificar la instrucción y ejecutarla. Por el contrario, la representación en *lenguaje ensamblador* sí está pensada para la mayor comodidad del programador. Podría argüirse que LD r, r' debería, en realidad, interpretarse como "transferir el contenido de r a r'". No obstante, se ha decidido convencionalmente, es decir, de forma arbitraria, que signifique lo contrario, para conservar la compatibilidad con la representación binaria.

Ejercicio 2.1: *Escribir el código binario encargado de transferir el contenido del registro C al registro B. Consúltense los códigos correspondientes a C y B en la figura 2.16.*

CODIGO	REGISTRO
0 0 0	B
0 0 1	C
0 1 0	D
0 1 1	E
1 0 0	H
1 0 1	L
1 1 0	-(MEMORIA)
1 1 1	A

Figura 2.16
Códigos de los registros.

Otro ejemplo sencillo de instrucción de una palabra sería:

ADD A, r

que da lugar a la suma del contenido de un registro especificado (r) al acumulador A. Simbólicamente, la operación podría representarse mediante la expresión A = A + r. Como se verá en el capítulo 4, la representación binaria de esta instrucción es:

10000FFF

siendo FFF el registro que debe añadirse al acumulador. Los códigos de los registros aparecen en la figura 2.16.

Ejercicio 2.2: *¿Cuál sería el código binario de la instrucción encargada de sumar el contenido del registro D al acumulador?*

INSTRUCCIONES DE DOS PALABRAS

ADD A, n

Esta sencilla instrucción de dos palabras hace que se sume el contenido del segundo byte de la instrucción al acumulador. El contenido de la segunda palabra de la instrucción es lo que se llama un "literal", es decir, un dato que se trata como grupo de ocho bits sin ningún significado particular. Podría ser un carácter o un dato numérico, pero lo importante es que su naturaleza es irrelevante a efectos de la operación. El código de la instrucción es:

11000110 seguido por el byte de 8 bits "n"

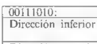
Se trata de una operación *inmediata*, cosa que, en la mayor parte de los lenguajes de programación, significa que la palabra o palabras siguientes contienen un fragmento de dato que no debe ser *interpretado* (en el sentido en que sí debe serlo un código de operación). De ello se deduce que la palabra o las dos palabras siguientes se tratan como *literales*.

La unidad de control está programada para que "sepa" cuántas palabras tiene cada instrucción, de manera que siempre toma y ejecuta el número correcto en cada caso. Sin embargo, cuanto mayor sea el número de palabras de la instrucción, tanto más complicado le resultará decodificarlas a la unidad de control.

INSTRUCCIONES DE TRES PALABRAS

LD A, (nn)

Esta instrucción necesita tres palabras. Significa: "cargar el acumulador con la dirección de la memoria especificada en los dos bytes siguientes de la instrucción". Como las direcciones tienen una longitud de 16 bits, ocupan dos palabras. La representación binaria sería:



00111010:	8 bits para el código de operación.
Dirección inferior:	8 bits para la parte inferior de la dirección.
Dirección superior:	8 bits para la parte superior de la dirección.

Ejecución de instrucciones dentro del Z80

Como ya hemos visto, todas las instrucciones se llevan a cabo en tres fases: TOMAR, DECODIFICAR y EJECUTAR. Daremos ahora algunas definiciones de interés. Cada una de las fases consume varios ciclos de reloj. El Z80 ejecuta cada una de ellas en uno o más ciclos lógicos que se llaman "ciclos de máquina"; el ciclo de máquina más breve dura tres ciclos de reloj.

El acceso a la memoria lleva tres ciclos para cualquier operando y cuatro de reloj para la fase tomar. Dado que lo primero que hay que hacer con cualquier instrucción es tomarla, las más rápidas necesitarán al menos cuatro ciclos de reloj, aunque la mayoría consumen más.

Los ciclos de máquina se designan M1, M2, etc., y cada uno de ellos consume tres o más ciclos de reloj o "estados", que se designan T1, T2, etc.

Figura 2.17
Tomar la instrucción: el PC dirige a la memoria.

FASE TOMAR (FETCH)

La fase de TOMAR la instrucción se lleva a cabo durante los primeros tres estados del ciclo de máquina M1, denominados T1, T2 y T3. Estos tres estados son comunes a todas las instrucciones del microprocesador, porque todas deben tomarse antes de pasar a su ejecución. El mecanismo de TOMAR es el siguiente:

T1: SALIDA del PC

El primer paso es la presentación a la memoria de la dirección de la instrucción siguiente, dirección que figura en el contador del programa (PC). Como ocurre en la fase tomar de todas las instrucciones, se empieza por colocar el contenido del PC en el *bus* de direcciones (véase la figura 2.17). En este momento se presenta una dirección a la memoria, donde es interpretada por el correspondiente decodificador de direcciones para seleccionar la posición de memoria correcta. Antes de que el contenido de

Figura 2.18
Incremento del PC.

el código de operación.
la parte inferior de la

la parte superior de la

l Z80

instrucciones se llevan a
DIFICAR y EJECUTAR.
interés. Cada una de las
Z80 ejecuta cada una de
ue se llaman "ciclos de
breve dura tres ciclos de

es ciclos para cualquier
ase tomar. Dado que lo
uier instrucción es tomar-
os cuatro ciclos de reloj.

M1. M2. etc.. y cada uno
reloj o "estados", que se

se lleva a cabo durante
máquina M1, denomina-
son comunes a todas las
que todas deben tomarse
anismo de TOMAR es el

a la memoria de la direc-
ón que figura en el conta-
en la fase tomar de todas
ar el contenido del PC en
2.17). En este momento se
donde es interpretada por
recciones para seleccionar
a de que el contenido de

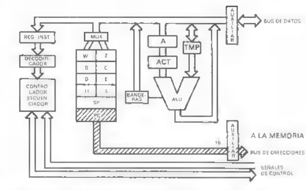


Figura 2.17
Tomar la instrucción: el PC se dirige a la memoria.

esa posición llegue a las patillas de salida de la memoria, conectadas al *bus* de datos, deben transcurrir varios cientos de nanosegundos (un nanosegundo, ns, es igual a 10^{-9} segundos). En diseño informático, es norma universal utilizar el tiempo de lectura en la memoria para realizar una operación dentro del microprocesador. Esta operación es el incremento del contador del programa:

T2: PC = PC + 1

Mientras se lee la memoria, el contenido del PC se incrementa en 1 (véase la figura 2.18). Al final del estado T2 el contenido de la memoria ya está disponible y puede transferirse dentro del microprocesador:

T3: INST en IR

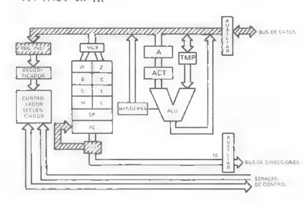


Figura 2.18
Incremento del PC.

FASES DECODIFICAR Y EJECUTAR

Durante el estado T3, la instrucción leída en la memoria se deposita en el *bus* de datos y se transfiere al registro de instrucción del Z80; la decodificación tiene lugar a partir de este punto.

Figura 2.20
Transferencia de C a ■.

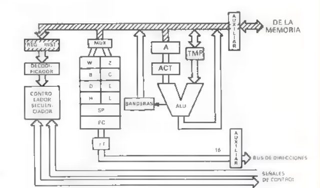


Figura 2.19
La instrucción llega a IR procedente de la memoria.

Hay que observar que M1 siempre debe alcanzar un estado T4; en efecto, una vez depositada la instrucción en IR durante T3, hay que *decodificarla y ejecutarla*, lo que exige, al menos, un nuevo estado de máquina T4.

Algunas instrucciones exigen otro estado adicional T5 de M1, que el procesador salta en la mayor parte de los casos. Si la instrucción exige dos o más ciclos M1, M2, etc., se produce directamente la transición desde el estado T4 de M1 al T1 de M2. Veremos a continuación un ejemplo, que estudiaremos con ayuda de las secuencias internas que refleja la tabla de la figura 2.27 (estas tablas no se han publicado todavía para el Z80, por lo que usamos en su lugar las correspondientes al 8080; dan una visión en profundidad de las fases que sigue la ejecución de una instrucción).

Figura 2.21
■ contenido de C se depo
en TMP.

LD D, C

Esta instrucción del Z80 corresponde a la MOV r1, r2 del 8080, que recoge en su primera línea la tabla de la figura 2.27.

Da la casualidad que el registro de destino de este ejemplo se llama también D. La transferencia se ilustra en la figura 2.20.

AR

n leída en la memoria se
ere al registro de instruc-
lugar a partir de este



debe alcanzar un estado
nstrucción en IR durante
o que exige, al menos, un

estado adicional T5 de
yor parte de los casos. Si
M1, M2, etc., se produce
ado T4 de M1 al T1 de
lo, que estudiaremos con
fleja la tabla de la figura
todavía para el Z80, por
pondientes al 8080; dan
que sigue la ejecución de

de a la MOV r1, r2 del
a tabla de la figura 2.27.
e destino de este ejemplo
e ilustra en la figura 2.20.

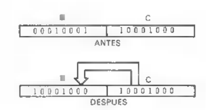


Figura 2.20
Transferencia de C a D.

La instrucción, que ya hemos descrito en la sección anterior, lleva el contenido del registro C, llamado "c", al registro D.

Los tres primeros estados del ciclo M1 se emplean en traer la instrucción de la memoria. Al término de T3 ya está en el registro de instrucción, IR, desde el que será decodificada (véase la figura 2.19).

Durante T4: (FFF) ► TMP.

El contenido de C se deposita en TMP (véase la figura 2.21).

Durante T5: (TMP) ► DDD.

El contenido de de TMP se deposita en D (véase la figura 2.22).

Figura 2.21
El contenido de C se deposita en TMP.

La fase de ejecución ya ha terminado. El contenido del registro C ha pasado al registro D, que constituía su destino de acuerdo con lo especificado, y de esta forma acaba la ejecución de la instrucción. Los ciclos de máquina M2, M3, M4 y M5 no son necesarios, y la operación completa se ha realizado en M1.

Es fácil calcular el tiempo real que tarda en ejecutarse la instrucción. En el Z80 normal, la duración de un estado es la del reloj: 500 ns. Esta instrucción requiere el transcurso de

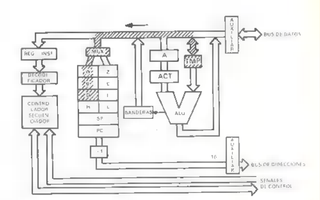


Figura 2.22
El contenido de TMP se deposita en D.

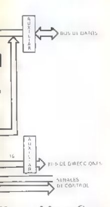
cinco estados, es decir, $5 \times 500 = 2\,500$ ns $= 2.5$ μs. Con un reloj de 400 ns, sería $5 \times 400 = 2\,000$ ns $= 2.0$ μs.

Pregunta: *¿Por qué esta instrucción necesita dos estados — T4 y T5 — para transferir el contenido de C a D en lugar de sólo uno? Primeramente lleva el contenido de C a TMP, y a continuación el de éste a D. ¿No sería más fácil pasar el contenido de C a D directamente y en un solo estado?*

Respuesta: Eso sería imposible, debido a la disposición de los registros internos del Z80. Todos los registros forman parte de una única memoria RAM de lectura/escritura construida dentro del microcircuito integrado que constituye el microprocesador. Como la RAM tiene una sola puerta, únicamente puede direccionarse o escogerse una palabra en un momento determinado, de tal manera que es imposible leer y escribir simultáneamente en dos posiciones diferentes de la memoria. Por eso hay que consumir dos ciclos de RAM, uno para leer los datos y almacenarlos en el registro temporal TMP y otro para escribirlos en el registro final de destino (D, en este caso). Es una insuficiencia de diseño común prácticamente a todos los microprocesadores monolíticos. Para resolver el problema sería necesario una RAM de doble puerta. De todas formas, la limitación no es intrínseca a los microprocesadores, y no se encuentra en los dispositivos "a rebanadas"*; es consecuencia de la tendencia dominante hacia el aumento de la densidad lógica de los circuitos integrados, y podría solucionarse en el futuro.

---

* *Bit slice:* "a rebanadas": diversos *chips* que forman una CPU (suele estar basada en *chips* de 4 bits; por ejemplo, la familia 2 900, formando UAL, de $n \times 4$ bits).



00 ns = 2,5 μs. Con un
ns = 2,0 μs.

esita dos estados T4 y
C a D en lugar de sólo
do de C a TMP, y a
sería más fácil pasar el
en un solo estado?

a la disposición de los
s registros forman parte
tura/escritura construida
que constituye el micro-
a sola puerta, únicamen-
una palabra en un mo-
que ■ imposible leer y
siciones diferentes de la
mir dos ciclos de RAM.
rlos en el registro tempo-
en el registro final de
insuficiencia de diseño
microprocesadores mono-
ería necesario una RAM
a limitación no es intrín-
no se encuentra en los
secuencia de la tendencia
a densidad lógica de los
cionarse en el futuro.

a forman una CPU (suele estar
ilia 2 900, formando UAL, de

EJERCICIO IMPORTANTE

Llegados ■ este punto, es muy recomendable que el lector repase la secuencia de esta sencilla instrucción antes de pasar a otras más complicadas. Vuelva para ello a la figura 2.14, reúna unos pocos "símbolos" pequeños —cerillas, *clips*, etc.— y desplácelos sobre la ilustración para simular el flujo de datos de los registros a los *buses*. Por ejemplo: deposite un símbolo en el PC; en T1, dicho símbolo recorrerá el *bus* direcciones hacia la memoria; continúe la simulación por el mismo procedimiento hasta que considere que domina los movimientos que tienen lugar entre *buses* y registros; en ese momento estará en condiciones de pasar a las siguientes instrucciones, que serán cada vez más complejas.

ADD A, r

La instrucción significa: «sumar el contenido del registro r (especificado mediante un código binario FFF) al acumulador (A) y depositar el resultado en dicho acumulador". Se trata de una instrucción *implícita*, es decir, que no hace referencia explícita a un segundo registro, sino sólo ■ uno llamado r, lo que implica que el otro afectado por la operación es el acumulador. Este, cuando se usa en una instrucción implícita, se toma como origen y como destino. Como resultado de la operación, los datos se almacenan en el acumulador. La ventaja de esta instrucción implícita es que el código de operación completo sólo tiene ocho bits de longitud y no necesita más que un campo de registro de tres bits para la especificación de r. Se trata, pues, de una forma rápida de sumar.

El sistema dispone de otras instrucciones implícitas que se refieren a registros especializados. Son ejemplos de instrucciones de esta clase las PUSH (empujar) y POP (extraer), que movilizan información entre el extremo superior de la pila y el acumulador al tiempo que actualizan el puntero de la pila (SP). Implícitamente manipulan el registro SP.

A continuación analizaremos en detalle la ejecución de ADD A, r. La instrucción precisa dos ciclos de máquina, M1 y M2. Como es usual, durante los tres primeros estados de M1 se trae la instrucción de la memoria y se deposita en el registro IR. Al principio de T4 se decodifica y puede ya ejecutarse. Supondremos aquí que el registro B se suma al acumulador. El código de la operación es 10000000 (el código del registro B es 000). La instrucción 8080 equivalente a ésta es ADD r.

T4: (FFF) ► TMP; (A) ► ACT

Como se ve, tienen lugar simultáneamente dos transferencias. Primero se transfiere a TMP el registro fuente especificado (B en este caso), lo que significa que se coloca a la entrada de la ALU (véase la figura 2.23). Al mismo tiempo se transfiere el contenido del acumulador al acumulador temporal ACT. Si observa la figura 2.23, comprenderá que los dos movimientos pueden realizarse en paralelo, porque siguen caminos diferentes dentro del sistema. El paso de B a TMP sigue el *bus* interno de datos, mientras que el de A a ACT se efectúa a través de un breve camino interior independiente del mencionado *bus* de datos. La realización simultánea de los dos movimientos supone ahorro de tiempo. En este momento las entradas derecha e izquierda de la ALU se encuentran correctamente determinadas; la primera, por el contenido del registro B, y la segunda, por el del acumulador. El siguiente paso es, pues, la adición. Cabría esperar que se llevase a cabo durante el estado T5 de M1, pero dicho estado no se utiliza, y la suma queda sin realizar. Se pasa al ciclo M2, pero durante el estado T1 tampoco ocurre nada. La adición no tiene lugar hasta el estado T2 de M2 (véase ADD r en la figura 2.27):

T2 de M2: (ACT) + (TMP) ► A

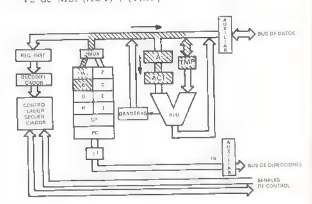


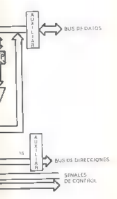
Figura 2.23
Dos transferencias que ocurren simultáneamente.

Figura 2.24
Final de ADD r.

El contenido de ACT se suma al de TMP, y el resultado se deposita en el acumulador (figura 2.24). La operación ya está completa.

Pregunta: *¿Por qué se difiere el término de la adición hasta el estado T2 del ciclo de máquina M2 en lugar de ejecutarla en el estado T5 de M1?* (Es difícil responder a esta pregunta sin

eamente dos transferen-
gistro fuente especificado
coloca a la entrada de la
tiempo se transfiere el
dor temporal ACT. Si
ue los dos movimientos
iguen caminos diferentes
P sigue el *bus* interno de
efectúa a través de un
del mencionado *bus* de
dos movimientos supone
las entradas derecha e
orrectamente determina-
registro B, y la segunda.
aso es, pues, la adición.
durante el estado T5 de
, y la suma queda sin
nte el estado T1 tampo-
gar hasta el estado T2 de



le TMP, y el resultado se
24). La operación ya está

no de la adición hasta el
2 en lugar de ejecutarla en
onder a esta pregunta sin

tener ciertos conocimientos sobre diseño de CPU; básicamente, se trata de una técnica de diseño de CPU sincronizada con el reloj. Trataremos a continuación de explicar lo que ocurre.)

Respuesta: Es un "truco" de diseño, común a casi todas las CPU, que se llama "solapamiento toma/ejecución". En esencia, se trata de lo siguiente: como se observa en la figura 2.23, la ejecución de la adición propiamente dicha sólo precisa de la ALU y del *bus* de datos y, en particular, no necesita acceder a la RAM (bloque de registros). La unidad de control "sabe" que los tres estados ejecutados a continuación de cualquier instrucción son siempre T1, T2 y T3 del ciclo de máquina M1 de la instrucción siguiente. Al repasar la ejecución de estos tres estados se aprecia que, para ello, basta acceder al contador del programa PC y utilizar el *bus* de direcciones. Acceder al contador del programa supone acceder a los registros de RAM (esto explica por qué no puede recurrirse a la misma idea en la instrucción LD, r, r'). Por tanto, es posible trabajar simultáneamente en las zonas sombreadas de las figuras 2.17 y 2.24.

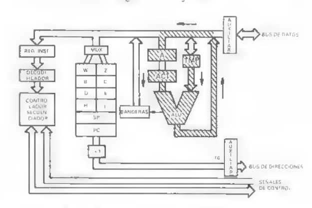


Figura 2.24
Final de ADD r.

El *bus* de datos se utiliza en T1 de M1 para transportar información sobre el estado, pero no puede utilizarse para la adición que deseamos ejecutar. Por ello es preciso esperar hasta el T2 antes de sumar, y eso es lo que ocurre en la tabla. La ventaja del mecanismo que acabamos de explicar la veremos a continuación: supongamos que procedemos de lo que sería la forma normal y ejecutamos la suma durante el estado T5 del ciclo de máquina M1; la duración de la instrucción ADD sería

de esta forma: 5 × 500 = 2 500 ns. Con el solapamiento, la instrucción siguiente comienza al término del estado T4, y la "astuta" unidad de control aprovechará el estado T2 de esta segunda instrucción para terminar la suma, pero sin que ello afecte a dicha siguiente instrucción. En la tabla, T2 se presenta como parte de M2, que, conceptualmente, es el segundo ciclo de máquina de la adición, aunque en realidad T2 pertenece al ciclo M1 de la instrucción siguiente. Para el programador, la instrucción ADD consume únicamente cuatro estados, es decir, 4 × 500 = 2 000 ns, en lugar de los 2 500 que emplearía el tratamiento convencional. La velocidad ha mejorado en 500 ns, equivalentes a un 20 por 100.

La técnica del solapamiento se describe en la figura 2.25, y se utiliza siempre que mediante ella sea posible incrementar la velocidad de trabajo aparente del microprocesador. Naturalmente, no en todos los casos se puede solapar, porque los *buses* y dispositivos necesarios han de estar disponibles, sin que ello suponga conflicto. La unidad de control "sabe" en todo momento si el solapamiento es o no posible.

Figura 2.25
Solapamiento TOMAR-EJECUTAR durante T1-T2.

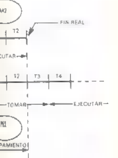
Pregunta: *¿Sería posible llevar más lejos este recurso y utilizar también el estado T3 de M2 para ejecutar una instrucción más larga?*

Para clarificar el mecanismo interno de secuenciación es aconsejable examinar la figura 2.27, que recoge en detalle la ejecución de las instrucciones del 8080. El Z80 tiene todas las instrucciones del 8080 y algunas más. Si se ha reproducido aquí es porque contribuye a esclarecer el funcionamiento interno de este microprocesador. Los apéndices F y G recogen las equivalencias entre el Z80 y el 8080. Pasemos ahora a estudiar una instrucción más compleja:

Figura 2.26
Abreviaturas Intel.

Con el solapamiento, la
ino del estado T4, y la
rá el estado T2 de esta
suma, pero sin que ello
la tabla, T2 se presenta
te, es el segundo ciclo de
dad T2 pertenece al ciclo
programador, la instruc-
atro estados, es decir,
2 500 que emplearía el
ha mejorado en 500 ns.

ribe en la figura 2.25, y
a posible incrementar la
croprocesador. Natural-
solapar, porque los *buses*
disponibles, sin que ello
rol "sabe" en todo mo-
sible.



os este recurso y utilizar
cutar una instrucción más

rno de secuenciación es
que recoge en detalle la
). El Z80 tiene todas las
Si se ha reproducido aquí
uncionamiento interno de
y G recogen las equiva-
os ahora a estudiar una

ADD A, (HL)

El código de operación es 10000110. La instrucción significa "sumar al acumulador el contenido de la posición de memoria (HL)". Esta posición se especifica de una forma un tanto extraña, porque es aquella cuya dirección está contenida en los registros H y L. La instrucción supone que estos dos registros

NOTAS:

1 El primer ciclo de memoria (M1) siempre es tomar una instrucción; durante el mismo se toma el primer (a veces, el único) byte del código de operación.

2 Si la entrada READY de la memoria no está en alto durante T2 de cada ciclo de memoria, el procesador entrará en estado de espera (TW) hasta que se detecte en alto.

3. Los estados T4 y T5 están disponibles, si son necesarios, para operaciones completamente internas a la CPU. El contenido del *bus* interno durante T4 y T5 está a disposición del *bus* de datos, aunque esto sólo sirve con fines de verificación. "X" significa que el estado está presente, pero sólo se utiliza en operaciones internas, como la decodificación de instrucciones.

4. Sólo pueden especificarse pares de registros pr = B (registros B y C) o pr = D (registros D y E).

5. Estos estados se saltan.

6. Subciclos de lectura en memoria, pueden leerse palabras de instrucción o de dato.

7 Subciclo de escritura en memoria.

8 La señal READY no es necesaria durante los subciclos segundo y tercero (M2 y M3). La señal HOLD se acepta durante M2 y M3. La señal SYNC no se genera durante M2 y M3. Durante la ejecución de DAD hacen falta M2 y M3 para la suma de un par de registros internos, no se referencia la memoria.

9 Los resultados de estas instrucciones aritméticas, lógicas o de rotación no se mueven al acumulador (A) hasta el estado T2 del siguiente ciclo de instrucción, de manera que A se carga mientras se toma la instrucción siguiente. Este solapamiento de operaciones acelera la velocidad de proceso.

10 Si el valor de los 4 bits menos significativos del acumulador es superior a 9 o si el bit auxiliar de arrastre se ha fijado, se suma 6 al acumulador. Si el valor de los 4 bits más significativos del acumulador es ahora superior a 9 o si el bit de arrastre se ha fijado, se suma 6 a los 4 bits más significativos del acumulador.

11 Esto representa el primer subciclo (traer la instrucción) del siguiente ciclo de instrucción.

12. Si se satisface la condición, el contenido del par de registros WZ se lleva a las líneas de salida (A_{0-15}) en lugar del contenido del contador del programa PC.

13. Si la condición no se satisface, se saltan los subciclos M4 y M5, el procesador pasa inmediatamente a la fase traer (M1) del siguiente ciclo de instrucción.

14. Si la condición no se satisface, se saltan los subciclos M2 y M3; el procesador pasa inmediatamente a la fase traer (M1) del siguiente ciclo de instrucción.

15. Subciclo de lectura de la pila.

16 Subciclo de escritura de la pila.

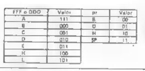
17. CONDICION — CCC

CONDICION	CCC
NZ – no cero (Z = 0)	000
Z – cero (Z = 1)	001
NC – sin acarreo (CY = 0)	010
C – acarreo (CY = 1)	011
PO – paridad impar (P = 0)	100
PE – paridad par (P = 1)	101
P – más (S = 0)	110
M – menos (S = 1)	111

18 Subciclo E/S, el código de selección de la puerta de E/S de 8 bits se duplica en las líneas de dirección 0-7 (A_{0-7}) y 8-15 (A_{8-15}).

19 Subciclo de salida.

20 El procesador permanecerá inactivo en estado de detención hasta que acepte una indicación de interrupción, reinicio (RESET) o mantenimiento (HOLD). Si se acepta esta última, la CPU pasa a función de mantener y, al final de la misma, de nuevo a estado de detención. Si se acepta el reinicio, el procesador empieza la ejecución por la posición de memoria cero. Tras aceptar una interrupción, el procesador ejecuta la instrucción introducida en el *bus* de datos (por lo general, una instrucción de vuelta a cero).



FFF o DDD	Valor	pr	Valor
A	111	B	00
B	000	D	01
C	001	H	10
D	010	SP	11
E	011		
H	100		
L	101		

Por cortesía de Intel Corporation

Figura 2.26
Abreviaturas Intel.

especiales (**HL**) se han cargado antes de la ejecución de la misma, de manera que sus contenidos de 16 bits especificarán la dirección de la memoria en que residen los datos. Estos datos se sumarán acto seguido al acumulador, que será el lugar en que se deposite también el resultado.

Esta instrucción se ha creado para garantizar la compatibilidad entre el antiguo 8008 y su sucesor, el 8080. El 8008 no disponía de direccionamiento directo de la memoria, por lo que para acceder a ésta se cargaban dos registros, H y L, y se ejecutaba una instrucción que se refiriese a ellos, como ADD A, (HL). Hay que insistir en que el 8080 y el Z80 no tienen la limitación de direccionamiento de la memoria del 8008, sino que cuentan con acceso directo a la misma; por tanto, la

MNEMONIC	OP CODE		M1[1]					M2		
	D7 D6 D5 D4	D3 D2 D1 D0	T1	T2[2]	T3	T4	T5	T1	T2[2]	T3
MOV r1, r2	[illegible]	[illegible]	PC OUT STATUS	PC = PC + 1	INST→TMP/IR	(SSS)→TMP	(TMP)→DDD			
MOV r, M	[illegible]	[illegible]				X[3]		HL OUT STATUS[6]	DATA	→DDD
MOV M, r	[illegible]	[illegible]				(SSS)→TMP		HL OUT STATUS[7]	(TMP)	→DATA BUS
SPHL	[illegible]	[illegible]				(HL)	SP			
MVI r, data	[illegible]	[illegible]				X		PC OUT STATUS[6]	B2	→DDD
MVI M, data	[illegible]	[illegible]				X			B2	→TMP
LXI rp, data	[illegible]	[illegible]				X			PC = PC + 1 B2	→r1
LDA addr	[illegible]	[illegible]				X			PC = PC + 1 B2	→Z
STA addr	[illegible]	[illegible]				X			PC = PC + 1 B2	→Z
LHLD addr	[illegible]	[illegible]				X			PC = PC + 1 B2	→Z
SHLD addr	[illegible]	[illegible]				X		PC OUT STATUS[6]	PC = PC + 1 B2	→Z
LDAX rp[4]	[illegible]	[illegible]				X		rp OUT STATUS[6]	DATA	→A
STAX rp[4]	[illegible]	[illegible]				X		rp OUT STATUS[7]	(A)	→DATA BUS
XCHG	[illegible]	[illegible]				(HL)↔(DE)				
ADD r	[illegible]	[illegible]				(SSS)→TMP (A)→ACT			(ACT)+(TMP)→A	
ADD M	[illegible]	[illegible]				(A)→ACT		HL OUT STATUS[6]	DATA	→TMP
ADI data	[illegible]	[illegible]				(A)→ACT		PC OUT STATUS[6]	PC = PC + 1 B2	→TMP
ADC r	[illegible]	[illegible]				(SSS)→TMP (A)→ACT		[9]	(ACT)+(TMP)+CY→A	
ADC M	[illegible]	[illegible]				(A)→ACT		HL OUT STATUS[6]	DATA	→TMP
ACI data	[illegible]	[illegible]				(A)→ACT		PC OUT STATUS[6]	PC = PC + 1 B2	→TMP
SUB r	[illegible]	[illegible]				(SSS)→TMP (A)→ACT		[9]	(ACT)−(TMP)→A	
SUB M	[illegible]	[illegible]				(A)→ACT		HL OUT STATUS[6]	DATA	→TMP
SUI data	[illegible]	[illegible]				(A)→ACT		PC OUT STATUS[6]	PC = PC + 1 B2	→TMP
SBB r	[illegible]	[illegible]				(SSS)→TMP (A)→ACT		[9]	(ACT)−(TMP)−CY→A	
SBB M	[illegible]	[illegible]				(A)→ACT		HL OUT STATUS[6]	DATA	→TMP
SBI data	[illegible]	[illegible]				(A)→ACT		PC OUT STATUS[6]	PC = PC + 1 B2	→TMP
INR r	[illegible]	[illegible]				(DDD)→TMP (TMP)+1→ALU	ALU→DDD			
INR M	[illegible]	[illegible]				X		HL OUT STATUS[6]	DATA (TMP)+1	→TMP →ALU
DCR r	[illegible]	[illegible]				(DDD)→TMP (TMP)−1→ALU	ALU→DDD			
DCR M	[illegible]	[illegible]				X		HL OUT STATUS[6]	DATA (TMP)−1	→TMP →ALU
INX rp	[illegible]	[illegible]				(RP)+1	RP			
DCX rp	[illegible]	[illegible]				(RP)−1	RP			
DAD rp[8]	[illegible]	[illegible]				X		(rl)→ACT	(L)→TMP (ACT)+(TMP)→ALU	ALU→L, CY
DAA	[illegible]	[illegible]				DAA→A, FLAGS[10]				
ANA r	[illegible]	[illegible]				(SSS)→TMP (A)→ACT		[9]	(ACT)+(TMP)→A	
ANA M	[illegible]	[illegible]	PC OUT STATUS	PC = PC + 1	INST→TMP/IR	(A)→ACT		HL OUT STATUS[6]	DATA	→TMP

Figura 2.27
Formatos de las instrucciones Intel.

Figura 2.27
Formatos de las instrucc
Intel (continuación).

posibilidad de trabajar con los registros H y L se convierte en una ventaja, y no en un inconveniente, como era en el 8008.

Sigamos ahora su ejecución (la instrucción se llama ADD M en el 8080, y ocupa el número 16 en la figura 2.27). Como es habitual, se utilizan los estados T1, T2 y T3 de M1 para tomar la instrucción. Durante el estado T4 se transfiere el contenido del acumulador a un registro auxiliar ACT, y se determina la entrada izquierda de la ALU.

Ahora es preciso acceder a la memoria para obtener el segundo byte de datos que debe sumarse al acumulador. La dirección de este byte está contenida en H y L, cuyo contenido, deberá, por tanto, transferirse al *bus* de direcciones y, mediante éste, a la memoria.

Figura 2.27
Formatos de las instrucciones Intel (continuación).

Durante el ciclo de máquina M2 se lee HL y su contenido se deposita en el *bus* de direcciones exactamente igual que en otras instrucciones se depositaba el contenido del PC. Ya se ha indicado que durante T1 se lleva el *estado* al *bus* de datos, aunque aquí no se hará uso del mismo. Simplificando las cosas, hacen falta dos estados: uno, para que la memoria lea los datos, y otro, para que los datos pasen al TMP de la entrada derecha de la ALU.

Ya están, pues, determinadas las dos entradas de la ALU. La situación es idéntica a la que ya hemos visto en la anterior instrucción, ADD A, r; por tanto, no hay más que efectuar la suma. Se recurre también a la técnica de solapamiento traer/ejecutar y, en lugar de hacer la suma en el estado T4 de M2, se

MNEMONIC	OP CODE		M1					M2		
	$D_7D_6D_5D_4$	$D_3D_2D_1D_0$	T1	T2	T3	T4	T5	T1	T2	T3
ANI data	1 1 1 0	0 1 1 0	PC OUT STATUS	PC = PC + 1	INST→TMP/IR	(A)→ACT		PC OUT STATUS	PC = PC + 1 B2	→TMP
XRA r	1 0 1 0	1 S S S				(A)→ACT (SSS)→TMP		[9]	(ACT)+(TMP)→A	
XRA M	1 0 1 0	1 1 1 0				(A)→ACT		HL OUT STATUS	DATA	→TMP
XRI data	1 1 1 0	1 1 1 0				(A)→ACT		PC OUT STATUS	PC = PC + 1 B2	→TMP
ORA r	1 0 1 1	0 S S S				(A)→ACT (SSS)→TMP		[9]	(ACT)+(TMP)→A	
ORA M	1 0 1 1	0 1 1 0				(A)→ACT		HL OUT STATUS	DATA	→TMP
ORI data	1 1 1 1	0 1 1 0				(A)→ACT		PC OUT STATUS	PC = PC + 1 B2	→TMP
CMP r	1 0 1 1	1 S S S				(A)→ACT (SSS)→TMP		[9]	(ACT)-(TMP), FLAGS	
CMP M	1 0 1 1	1 1 1 0				(A)→ACT		HL OUT STATUS	DATA	→TMP
CPI data	1 1 1 1	1 1 1 0				(A)→ACT		PC OUT STATUS	PC = PC + 1 B2	→TMP
RLC	0 0 0 0	0 1 1 1				(A)→ALU ROTATE		[9]	(ALU)→A, CY	
RRC	0 0 0 0	1 1 1 1				(A)→ALU ROTATE		[9]	(ALU)→A, CY	
RAL	0 0 0 1	0 1 1 1				(A), CY→ALU ROTATE		[9]	(ALU)→A, CY	
RAR	0 0 0 1	1 1 1 1				(A), CY→ALU ROTATE		[9]	(ALU)→A, CY	
CMA	0 0 1 0	1 1 1 1				(Ā)→A				
CMC	0 0 1 1	1 1 1 1				$\overline{CY}$→CY				
STC	0 0 1 1	0 1 1 1				1→CY				
JMP addr	1 1 0 0	0 0 1 1				X		PC OUT STATUS	PC = PC + 1 B2	→Z
J cond addr [17]	1 1 C C	C 0 1 0				JUDGE CONDITION		PC OUT STATUS	PC = PC + 1 B2	→Z
CALL addr	1 1 0 0	1 1 0 1				SP = SP - 1		PC OUT STATUS	PC = PC + 1 B2	→Z
C cond addr [17]	1 1 C C	C 1 0 0				JUDGE CONDITION IF TRUE, SP = SP - 1		PC OUT STATUS	PC = PC + 1 B2	→Z
RET	1 1 0 0	1 0 0 1				X		SP OUT STATUS	SP = SP + 1 DATA	→Z
R cond addr [17]	1 1 C C	C 0 0 0			INST→TMP/IR	JUDGE CONDITION		SP OUT STATUS	SP = SP + 1 DATA	→Z
RST n	1 1 N N	N 1 1 1			φ→W INST→TMP/IR	SP = SP - 1		SP OUT STATUS	SP = SP - 1 PCH	→DATA BUS
PCHL	1 1 1 0	1 0 0 1			INST→TMP/IR	(HL)	→PC			
PUSH rp	1 1 R P	0 1 0 1				SP = SP - 1		SP OUT STATUS	SP = SP - 1 (rh)	→DATA BUS
PUSH PSW	1 1 1 1	0 1 0 1				SP = SP - 1		SP OUT STATUS	SP = SP - 1 (A)	→DATA BUS
POP rp	1 1 R P	0 0 0 1				X		SP OUT STATUS	SP = SP + 1 DATA	→rl
POP PSW	1 1 1 1	0 0 0 1				X		SP OUT STATUS	SP = SP + 1 DATA	→FLAGS
XTHL	1 1 1 0	0 0 1 1				X		SP OUT STATUS	SP = SP + 1 DATA	→Z
IN port	1 1 0 1	1 0 1 1				X		PC OUT STATUS	PC = PC + 1 B2	→Z, W
OUT port	1 1 0 1	0 0 1 1				X		PC OUT STATUS	PC = PC + 1 B2	→Z, W
EI	1 1 1 1	1 0 1 1				SET INTE F/F				
DI	1 1 1 1	0 0 1 1				RESET INTE F/F				
HLT	0 1 1 1	0 1 1 0				X		PC OUT STATUS	HALT MODE [20]	
NOP	0 0 0 0	0 0 0 0	PC OUT STATUS	PC = PC + 1	INST→TMP/IR	X				

Figura 2.27
Formatos de las instrucciones Intel (continuación).

Figura 2.27
Formatos de las instrucci
Intel (continuación).

e lee HL y su contenido
xactamente igual que en
ntenido del PC. Ya se ha
estado al *bus* de datos.
. Simplificando las cosas,
la memoria lea los datos.
MP de la entrada derecha

os entradas de la ALU.
emos visto en la anterior
hay más que efectuar la
e solapamiento traer/eje-
el estado T4 de M2, se

	M	
T1	T2[3]	T3
PC OUT STATUS	PC = PC + 1 B2	→TMP
9	(ACT)+(TMP)→A	
HL OUT STATUS	DATA	→TMP
PC OUT STATUS	PC = PC + 1 B2	→TMP
9	(ACT)+(TMP)→A	
HL OUT STATUS	DATA	→TMP
PC OUT STATUS	PC = PC + 1 B2	→TMP
9	(ACT)-(TMP), FLAGS	
HL OUT STATUS	DATA	→TMP
PC OUT STATUS	PC = PC + 1 B2	→TMP
9	ALU→A, CY	
9	ALU→A, CY	
9	ALU→A, CY	
9	ALU→A, CY	
PC OUT STATUS	PC = PC + 1 B2	→Z
PC OUT STATUS	PC = PC + 1 B2	→Z
PC OUT STATUS	PC = PC + 1 B2	→Z
PC OUT STATUS	PC = PC + 1 B2	→Z
SP OUT STATUS	SP = SP + 1 DATA	→Z
SP OUT STATUS	SP = SP + 1 DATA	→Z
SP OUT STATUS	SP = SP - 1 (PCH)	→DATA BUS
SP OUT STATUS	SP = SP - 1 (rr)	→DATA BUS
SP OUT STATUS	SP = SP - 1 (rr)	→DATA BUS
SP OUT STATUS	SP = SP + 1 DATA	→rr
SP OUT STATUS	SP = SP + 1 DATA	→SLAGS
SP OUT STATUS	SP = SP + 1 DATA	→Z
PC OUT STATUS	PC = PC + 1 B2	→Z
PC OUT STATUS	PC = PC + 1 B2	→Z
PC OUT STATUS	HALT MODE[2]	

retrasa la ejecución hasta el T2 de M3. Como se ve en la figura 2.27, durante T2: ACT + TMP → A. Por fin se realiza la suma de los contenidos de ACT y TMP, y el resultado se deposita en el acumulador A.

Pregunta: *¿Cuál es el tiempo aparente de ejecución de la instrucción para el programador? Si se utilizase un reloj de 2.5 Mhz, ¿sería de 3.6 µs o de 2.8 µs?*

Examinaremos ahora una instrucción más complicada con direccionamiento directo de la memoria que utiliza dos registros invisibles, W y Z:

M3			M4			M5						
T1	T2[3]	T3	T1	T2[3]	T3	T1	T2[3]	T3	T4	T5		
9	(ACT)+(TMP)→A											
9	(ACT)+(TMP)→A											
9	(ACT)+(TMP)→A											
9	(ACT)+(TMP)→A											
9	(ACT)+(TMP)→A											
9	(ACT)+(TMP), FLAGS											
9	(ACT)+(TMP), FLAGS											
PC OUT STATUS	PC = PC + 1 B3	→W									WZ OUT STATUS	(WZ) + 1 → PC
PC OUT STATUS	PC = PC + 1 B3	→W									WZ OUT STATUS	(WZ) + 1 → PC
PC OUT STATUS	PC = PC + 1 B3	→W	SP OUT STATUS	(PCH) SP = SP - 1	→DATA BUS	SP OUT STATUS	(PCL) → DATA BUS				WZ OUT STATUS	(WZ) + 1 → PC
PC OUT STATUS	PC = PC + 1 B3	→W[3]	SP OUT STATUS	(PCH) SP = SP - 1	→DATA BUS	SP OUT STATUS	(PCL) → DATA BUS				WZ OUT STATUS	(WZ) + 1 → PC
SP OUT STATUS	SP = SP + 1 DATA	→W									WZ OUT STATUS	(WZ) + 1 → PC
SP OUT STATUS	SP = SP + 1 DATA	→W									WZ OUT STATUS	(WZ) + 1 → PC
SP OUT STATUS	(TMP) = 00NNN000 (PCL)	→Z →DATA BUS									WZ OUT STATUS	(WZ) + 1 → PC
SP OUT STATUS	(rl)	→DATA BUS										
SP OUT STATUS	FLAGS	→DATA BUS										
SP OUT STATUS	SP = SP + 1 DATA	→rh										
SP OUT STATUS	SP = SP + 1 DATA	→A										
SP OUT STATUS	DATA	→W	SP OUT STATUS	(H)	→DATA BUS	SP OUT STATUS	(L)	→DATA BUS	WZ	→HL		
WZ OUT STATUS	DATA	→L										
WZ OUT STATUS	(A)	→DATA BUS										

Figura 2.27
Formatos de las instrucciones Intel (continuación).

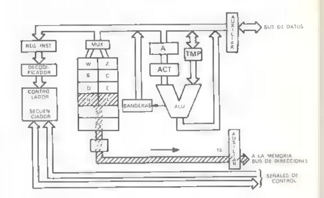


Figura 2.28
Transferencia del contenido de HL al *bus* de direcciones.

LD A, (nn)

El código de operación es 00111010. La instrucción equivalente del 8080 es LDA addr. Como es habitual, se emplean los estados T1, T2 y T3 de M1 para traer la instrucción de la memoria. También se utiliza T4, pero no puede describirse ningún resultado visible, porque durante ese estado se decodifica la instrucción. En ese momento, la unidad de control averigua que debe traer los siguientes dos bytes de la instrucción para obtener la dirección [redacted] partir de la que debe cargarse el acumulador. El efecto de esta instrucción [redacted] cargar el acumulador con los contenidos de memoria cuyas direcciones especifican los bytes 2 y 3 de la misma. Obsérvese que es preciso el estado T4 para *decodificar* la instrucción; esto podría considerarse una pérdida de tiempo, porque para esa labor basta una parte del estado, pero esa es la filosofía de la *lógica sincronizada por reloj*. Dado que internamente se utilizan microinstrucciones para llevar a cabo la decodificación y la ejecución, ése es el precio que hay que pagar [redacted] cambio de las ventajas de la microprogramación. La estructura de la instrucción aparece en la figura 2.29.

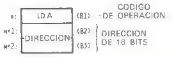
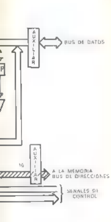


Figura 2.29
LD A, (DIRECCION) es una instrucción de tres palabras.



■. La instrucción equiva-
habitual, se emplean los
raer la instrucción de la
ro no puede describirse
te ese estado se decodifica
idad de control averigua
es de la instrucción para
debe cargarse el acumula-
gar el acumulador con los
nes especifican los bytes 2
reciso el estado T4 para
considerarse una pérdida
sta una parte del estado.
cronizada por reloj. Dado
strucciones para llevar ■
e es el precio que hay que
■ microprogramación. La
en la figura 2.29.

CODIGO
DE OPERACION
DIRECCION
DE 16 BITS

A continuación se toman los dos bytes siguientes de la instrucción, que especificarán una dirección (véase la figura 2.30).

El resultado de la instrucción se ilustra en las figuras 2.30 y 2.31 de esta misma página.

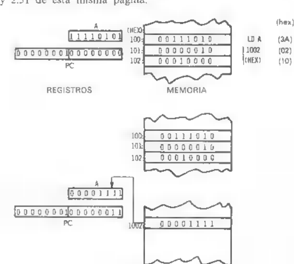


Figura 2.30
Situación previa a ■ ejecución de LD A.

Figura 2.31
Situación posterior a la ejecución de LD A.

La unidad de control (pero no el programador) tiene acceso a dos registros especiales internos del Z80: son los "W" y "Z", ilustrados en la figura 2.28.

Segundo ciclo de máquina M2: Como es habitual, los primeros dos estados, T1 y T2, se utilizan para tomar los contenidos de la posición de memoria PC. Durante T2 se incrementa el contador del programa. Hacia el final de este estado, los datos de la memoria quedan disponibles, y aparecen en el *bus* de datos. Al final de T3, la palabra que había sido tomada de la dirección de memoria PC (B2, segundo byte de la instrucción) está disponible en el *bus* de datos, que lo conduce a un registro temporal, Z: B2 ► Z (véase la figura 2.32).

Ciclo de máquina M3: Una vez más se deposita el PC en el *bus* de direcciones; se incrementa y, por último, se lee en la memoria, y se deposita en el registro W del microprocesador el tercer byte B3. En este punto, al final del estado T3 de M3, los registros internos W y Z contienen B2 y B3, es decir, la direc-

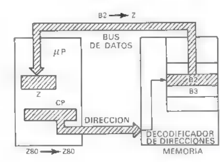


Figura 2.32
El segundo byte de la instrucción llega a Z.

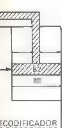
ción completa de 16 bits originalmente contenida en la memoria en forma de dos palabras tras la instrucción. La ejecución puede ya completarse: W y Z contienen una dirección que debe enviarse a la memoria para extraer el dato; estas operaciones se llevan a cabo en el siguiente ciclo de memoria.

Ciclo de máquina M4: W y Z envían a la memoria la dirección de 16 bits a través del *bus* de direcciones; al término del estado T2, quedan disponibles los datos correspondientes al contenido de la posición de memoria especificada, que se deposita en A al final del estado T3, con lo que concluye la ejecución de la instrucción.

Lo que acabamos de ver ilustra el uso de una *instrucción inmediata*, que precisa de tres bytes para almacenar una *dirección explícita* de dos bytes. Consume cuatro ciclos de memoria, porque necesita ir tres veces a ésta para tomar los tres bytes de las tres palabras y otra más para traer el dato especificado por la dirección. Se trata de una instrucción larga, pero, a la vez, es básica para cargar el acumulador con contenidos especificados que residen en una posición de memoria conocida. Obsérvese que la instrucción exige el empleo de los registros W y Z.

Pregunta: *¿Podría esta instrucción haber utilizado otros registros internos al sistema diferentes de W y Z?*

Respuesta: No. Si esta instrucción utilizase otros registros, los H y L por ejemplo, modificaría sus contenidos, que habrían perdido al término de la ejecución. En un programa se supone que una instrucción no modificará más registros que los que emplea explícitamente. Si la instrucción carga el acumulador, no ha de destruir el contenido de ningún otro registro; por ello es imprescindible crear dos registros adicionales, W y Z, para uso interno de la unidad de control.



contenida en la memoria
strucción. La ejecución
una dirección que debe
ato; estas operaciones se
memoria.
nvian a la memoria la
e direcciones: al término
latos correspondientes al
specificada, que se depo-
que concluye la ejecución

uso de una *instrucción*
ra almacenar una *direc-*
uatro ciclos de memoria,
a tomar los tres bytes de
el dato especificado por
larga, pero, a la vez, es
contenidos especificados
ria conocida. Obsérvese
los registros W y Z.

r utilizado otros registros
y Z?

izase otros registros, los
contenidos, que habrían
n. En un programa se
lificará más registros que
la instrucción carga el
ontenido de ningún otro
crear dos registros adi-
de la unidad de control.

Pregunta: *¿Podría haberse utilizado el PC en lugar de W y Z?*

Respuesta: De ninguna manera. Eso resultaría suicida (queda para el lector la demostración del porqué).

Veremos ahora un nuevo tipo de instrucción, llamada de *bifurcación* o *salto*, que modifica la secuencia de ejecución de las instrucciones del programa. Hasta ahora hemos supuesto que las instrucciones se ejecutaban unas tras otra, pero veremos que hay algunas que permiten al programador saltar a un punto del programa diferente o, en términos prácticos, saltar a otra zona de la memoria que alberga el programa o a otra dirección. Una instrucción de esta naturaleza es:

JP nn

La instrucción aparece en la línea 18 de la figura 2.27 como "JPM addr", y su ejecución puede seguirse en la misma tabla. Se trata también de una instrucción de tres palabras. La primera es el código de operación 11000011. Las dos siguientes contienen la dirección de 16 bits a la que debe saltarse. Conceptualmente, el efecto de la instrucción es sustituir el contenido del contador del programa por los 16 bits que siguen al código de operación "JUMP" (saltar). En la práctica, y por razones de eficacia, la cuestión se aborda de forma algo diferente.

Como antes, los tres primeros estados de M1 se emplean en tomar la instrucción. En T4 se decodifica ésta, y no se registra ningún otro suceso (X). Los dos siguientes ciclos de máquina se emplean en tomar los bytes B2 y B3 de la instrucción. En M2 se toma B2 y se deposita en el registro interno Z. Como en la suma, el procesador dará los dos pasos siguientes mientras realiza la siguiente operación de tomar. Dichos dos pasos se ejecutan en lugar de los T1 y T2 habituales de la instrucción siguiente, como veremos a continuación.

Esos dos pasos son: sacar WZ y (WZ) + 1 ► PC. En otras palabras, durante la siguiente operación de tomar se utiliza el contenido de WZ en lugar del contenido del PC. La unidad de control tendrá en cuenta que se está dando un salto, y ejecutará el principio de la instrucción siguiente de forma distinta.

El efecto de esos dos estados adicionales es el siguiente: la dirección depositada en el *bus* de direcciones del sistema será la contenida en W y Z, es decir, se tomará de la dirección contenida en W y Z, lo que, efectivamente, supone dar un *salto*.

Además, el contenido de WZ se incrementa en 1 y se deposita en el contador del programa, para que la siguiente instrucción se tome correctamente utilizando el PC de la forma habitual. El efecto es, pues, el buscado.

Pregunta: *¿Por qué se usan los registros intermedios W y Z en vez de cargar directamente el contenido del PC?*

Respuesta: El PC no puede usarse. Si cargásemos la parte inferior del mismo (PCL) con B2 en lugar de usar Z, destruiríamos el PC, y sería imposible tomar B3.

Pregunta: *¿Sería posible utilizar sólo Z en lugar de W y Z?*

Respuesta: Sí, pero el proceso sería más lento. Se podría cargar Z con B2, tomar a continuación B3 y depositarlo en la mitad superior del PC (PCH), pero ello obligaría a transferir Z al PCL antes de utilizar el contenido del PC, y el avance sería así más lento. Por eso se trabaja con W y Z. Además, y para ganar tiempo, W y Z no se transfieren al PC, sino que se llevan directamente al *bus* de direcciones para tomar la instrucción siguiente. Comprender este punto es crucial para comprender la ejecución eficaz de instrucciones dentro del microprocesador.

Pregunta (sólo para el lector con cierta experiencia): *¿Qué ocurriría si se produjese una interrupción al final de M3?* (Si la ejecución de la instrucción se suspendiera en ese punto, el contador del programa señalaría la instrucción siguiente al salto, y la dirección de éste, contenida en W y Z, se perdería.)

La averiguación de la respuesta constituirá un ejercicio provechoso para el lector con cierto nivel de conocimientos.

Las desripciones detalladas de instrucciones típicas que hemos visto habrán clarificado la función de los registros y de los *buses* internos. Una segunda lectura de todo lo anterior contribuirá a afianzar los conocimientos sobre las operaciones que tienen lugar en el interior del Z80.

Figura 2.33
Patillas de salida del μP Z80.

EL MICROCIRCUITO INTEGRADO Z80

Para completar este capítulo estudiaremos las señales del circuito integrado Z80. Conocer estas señales no es necesario para programar, y el lector no interesado por los dealles del

soporte físico puede saltarse esta parte con toda tranquilidad. La disposición de las patillas del Z80 se ilustra en la figura 2.33. El *bus* de direcciones y el de datos, a la derecha del dibujo, cumplen su función habitual, ya descrita al principio del capítulo. Estudiaremos aquí las señales del *bus* de control, que aparecen en el lado izquierdo de la misma figura.

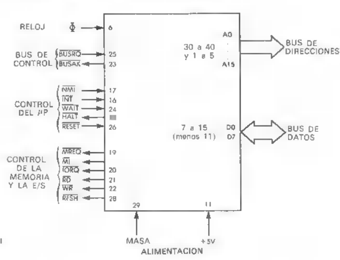


Figura 2.33
Patillas de salida del μP del Z80.

Esas señales de control se dividen en cuatro grupos, y comenzaremos su descripción por la parte superior del dibujo.

La entrada del reloj es Φ. El Z80 necesita una resistencia de 330 ohmios, que se conecta a las entradas Φ y a la toma de 5 voltios; a 4 MHz hace falta un amplificador externo para el reloj.

Las dos señales del *bus* de control, BUSRQ y BUSAK, sirven para desconectar el Z80 de sus *buses*, y son utilizadas, sobre todo, por el DMA (acceso directo a la memoria, *direct memory access*), aunque son igualmente accesibles por otros procesadores del sistema. BUSRQ es la señal de solicitud de *bus*; en respuesta a ella, el Z80 pone sus *bus* de direcciones y de datos y las señales de control de la salida de triple estado en situación de alta impedancia al término del ciclo de máquina en curso. BUSAK es la señal de reconocimiento emitida por el Z80 una vez que los *buses* han pasado a situación de alta impedancia.

Seis de las señales de control del Z80 están relacionadas con su estado interno o con el secuenciamiento:

INT y NMI son dos señales de interrupción. INT es la solicitud de interrupción habitual (las interrupciones se tratarán en el capítulo 6). A la línea INT pueden conectarse varios dispositivos de entrada y salida. Cuando se presenta en la misma una solicitud de interrupción —y si el biestable interno de permisión de interrupciones (IFF) se encuentra en el estado adecuado—, el Z80 la acepta (en el supuesto de que la entrada BUSRQ no esté activada); [redacted] continuación emitirá una señal de reconocimiento IORQ, que se produce durante el estado M1. El resto de la secuencia de acontecimientos se describe en el capítulo [redacted].

NMI es la interrupción no enmascarable, aceptada siempre por el Z80, que salta [redacted] la posición hexadecimal 0066 (suponiendo, igualmente, que BUSRQ no esté activada). Se describe en el capítulo 6.

WAIT (esperar) es la señal que se utiliza para sincronizar el Z80 con los dispositivos de memoria o de entrada y salida de baja velocidad. Cuando se activa, la señal indica que la memoria o el dispositivo no están todavía listos para la transferencia de datos; como respuesta, la CPU del Z80 entra en un estado especial de espera hasta que la señal WAIT se inactiva, momento en el que continúa la secuenciación normal.

HALT (alto) es la señal de reconocimiento que envía el Z80 después de haber ejecutado una instrucción HALT. En este estado, el Z80 espera una interrupción externa mientras continúa ejecutando ininterrumpidamente instrucciones NOP para refrescar la memoria.

RESET es la señal que habitualmente inicializa el μP. Pone a "0" el contador del programa y los registros I y R; incapacita el biestable de permisión de interrupciones y pone a "0" la función de interrupción. Normalmente se utiliza tras haber conectado la alimentación de la placa.

CONTROL DE LA MEMORIA Y DE LA E/S

El Z80 genera seis señales de control de la memoria y de los dispositivos de E/S.

MREQ es la señal de petición de memoria, que indica que la dirección presentada en el *bus* de direcciones es válida. A continuación puede efectuarse en la memoria una operación de lectura [redacted] escritura.

M1 es el ciclo de máquina 1 que corresponde a la fase tomar de una instrucción.

Resumen de

IORQ es la petición de entrada/salida. Indica que la dirección de E/S presentada en los bits 0-7 del *bus* de direcciones es válida. A continuación puede efectuarse una operación E/S de lectura o escritura. IORQ se emite también junto con M1 cuando el Z80 reconoce una interrupción: esta información puede ser aprovechada por microcircuitos externos para colocar el vector de respuesta ■ la interrupción en el *bus* de datos (las operaciones E/S normales nunca tienen lugar durante el estado M1; la combinación IORQ más M1 indica una situación de reconocimiento de interrupción).

RD es la señal de lectura (utilizada en combinación con MREQ ■ IOREQ), e indica que el Z80 está listo para leer el contenido del *bus* de datos en un registro interno. Puede utilizarla un circuito externo de memoria o de E/S para depositar datos en el *bus* de datos.

WR es la señal de escritura utilizada en combinación con MREQ o IOREQ. ■ indica que el *bus* de datos contiene un dato válido, listo para ser escrito en el dispositivo especificado.

RFSH es la señal de refresco. Cuando se activa, los siete bits inferiores del *bus* de direcciones contienen una dirección de refresco para memorias dinámicas. En ese momento se utiliza la señal MREQ para proceder al refresco leyendo la memoria.

Resumen del hardware

Con esto terminamos la descripción de la organización interna del Z80. Los detalles exactos del soporte físico no son importanes en el contexto en que nos estamos moviendo, pero si es necesario saber la función de cada uno de los registros, que debe conocerse perfectamente antes de pasar ■ los capítulos siguientes. Veremos a continuación el conjunto de instrucciones del Z80 y las técnicas básicas de programación de este microprocesador.



Té
de

Introducción

3

Técnicas básicas de programación

Introducción

El objetivo de este capítulo es enseñar las técnicas elementales necesarias para escribir programas con el Z80. Presentaremos aquí conceptos nuevos, como son los de organización de registros, bucles y subrutinas. Las técnicas de programación se centrarán exclusivamente en los recursos *internos* del Z80, es decir, en los registros. Desarrollaremos, además, programas reales, en particular programas aritméticos que servirán para ilustrar los conceptos expuestos y para utilizar instrucciones reales. Veremos así qué instrucciones hay que emplear para manipular la información entre la memoria y el μP, y dentro de esta última. En el capítulo siguiente analizaremos con todo detalle las instrucciones de que dispone el Z80. El capítulo 5 irá dedicado al estudio de las técnicas de direccionamiento, y el 6, al manejo de la información *fuera* del Z80 o, lo que es lo mismo, a las técnicas de entrada y salida.

En este capítulo aprenderemos, sobre todo, con la práctica. Al estudiar programas de complejidad creciente aprenderemos la función de las diferentes instrucciones y de los registros, y aplicaremos los conceptos ya desarrollados. No obstante, hay uno muy importante —el de técnicas de direccionamiento—

que, por su aparente complejidad, no abordaremos hasta el capítulo 5.

Tras esta breve introducción, pasaremos sin más a escribir algunos programas, empezando por los aritméticos. La figura 3.1 recoge el "modelo para el programador" de los registros del Z80.

GRUPO PRINCIPAL				GRUPO ALTERNATIVO		
(111)	A (acumulador)	F (banderas)		A'	F'	
(000)	B	C	(001)	B'	C'	registros generales
(010)	D	E	(011)	D'	E'	
(100)	H	L	(101)	H'	L'	

I (vector de int.)	R (refresco mem.)
IX	REGISTROS DE INDICE
IY	
SP (puntero de la pila)	
PC (contador del programa)	

Figura 3.1
Registros del Z80.

Programas aritméticos

Son programas aritméticos los de suma, resta, multiplicación y división. Los presentados aquí funcionarán con enteros, bien positivos binarios, bien en notación de complemento a dos, correspondiendo en este último caso el bit de la izquierda al signo (véase en el capítulo 1 la descripción de la notación en complemento a dos).

SUMA DE 8 BITS

Vamos a sumar dos operandos de 8 bits denominados OP1 y OP2, almacenados, respectivamente, en las direcciones de memoria ADR1 y ADR2. Llamaremos a la suma RES, y la almacenaremos en la dirección ADR3. Todo esto queda recogido en la figura 3.2. El programa encargado de realizar la suma es el siguiente:

Figura 3.2
Suma de 8 bits: RES = OP1 + OP2.

Instrucciones	*Comentarios*
LD A, (ADR1)	CARGAR OP1 EN A
LD HL, ADR2	CARGAR LA DIRECCION DE OP2 EN HL
ADD A, (HL)	SUMAR OP2 A OP1
LD (ADR3), A	DEJAR EL RESULTADO RES EN ADR3

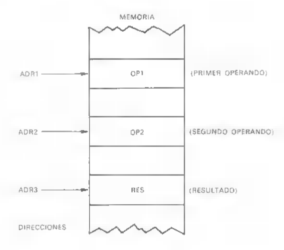


Figura 3.2
Suma de 8 bits: RES = OP1 + OP2.

Este es nuestro primer programa. Las instrucciones están a la izquierda, y los comentarios a las mismas, a la derecha. Examinémoslo con más detenimiento. Se trata de un programa de cuatro instrucciones (se llama *instrucción* a cada una de las líneas, y se ha expresado aquí de forma simbólica). El programa *ensamblador* traducirá cada una de esas instrucciones a uno, dos, tres o cuatro bytes, aunque este extremo no es de nuestra incumbencia.

En la primera línea se especifica que el contenido de ADR1 se cargue en el acumulador A. Como se ve en la figura 3.2, ese contenido es el primer operando "OP1"; por tanto, el resultado de la primera instrucción es la transferencia de OP1 desde la memoria al acumulador; el movimiento se ilustra en la figura 3.3. "ADR1" es una representación simbólica de la dirección real de 16 bits que se encuentra en la memoria. El símbolo ADR1 se definirá en otra parte el programa; podría, por ejemplo, definirse como igual a la dirección "100".

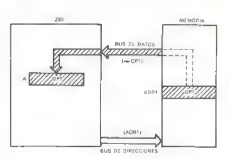


Figura 3.3
LD A, (ADR1): se carga OP1 de la memoria.

Esta instrucción de *carga* da lugar a una *operación de lectura* de la dirección 100 (véase la figura 3.3), cuyo contenido recorrerá el *bus* de datos hasta el acumulador, en el que queda depositado. Recordaremos del capítulo anterior que las operaciones lógicas y aritméticas actúan sobre el acumulador como uno de los operandos fuente (diríjase al mencionado capítulo para más detalles). Como queremos sumar los dos valores OP1 y OP2, debemos empezar por cargar OP1 en el acumulador; hecho esto, podremos sumar los contenidos del mismo, es decir, OP1 y OP2. El campo de la derecha de la instrucción se llama campo de *comentario*. El programa ensamblador lo ignora durante la traducción, pero de todas formas se incluye en el programa por razones de legibilidad. Para entender lo que hace el programa, es de capital importancia usar buenos comentarios (es lo que se llama *documentar* un programa).

En este caso el comentario se explica a sí mismo: el valor de OP1, que se encuentra en la dirección ADR1, se carga en el acumulador A.

El resultado de esta primera instrucción se ilustra en la figura 3.3. La segunda instrucción del programa es:

LD HL, ADR2

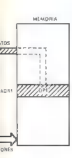
Obliga a "cargar ADR2 en los registros H y L". Para leer en la memoria el segundo operando OP2, antes debemos colocar su dirección en un par de registros del Z80, como H y L. A continuación podemos sumar el contenidio de la posición de memoria cuya dirección está en H y L al acumulador.

ADD A, (HL)

Como se ve en la figura 3.2, el contenido de la posición de memoria ADR2 es OP2, nuestro segundo operando. El conteni-

Figura 3.4
ADD A, (HL).

Figura 3.5
LD (ADR3), A (guardar el mulador en la memoria).



gar a una *operación de* gura 3.3), cuyo contenido mulador, en el que queda o anterior que las opera- bre el acumulador como e al mencionado capítulo umar los dos valores OP1 OP1 en el acumulador; nidos del mismo, es decir, de la instrucción se llama ensamblador lo ignora formas se incluye en el Para entender lo que hace usar buenos comentarios rograma).

ca a sí mismo; el valor de ón ADR1, se carga en el

trucción se ilustra en la el programa es:

gistros H y L". Para leer P2, antes debemos colocar del Z80, como H y L. A enido de la posición de L al acumulador.

ntenido de la posición de ndo operando. El conteni-

do del acumulador es en este momento OP1, el primer operando. Como resultado de la ejecución de esta instrucción, se toma OP2 de la memoria y se suma a OP1, como ilustra la figura 3.4.

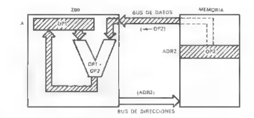


Figura 3.4
ADD A, (HL).

La suma se deposita en el acumulador. El lector recordará que, en el caso del Z80, el resultado de las operaciones aritméticas se deposita en el acumulador. En otros procesadores puede depositarse en otros registros o volver a la memoria.

La suma de OP1 y OP2 está, pues, en el acumulador. Para completar el programa no tenemos más que transferir el contenido de aquél a la posición de memoria ADR3 para almacenar el resultado en la posición especificada. Esta operación es la que realiza la cuarta instrucción del programa:

LD (ADR3), A

Esta instrucción carga el contenido de A en la dirección ADR3. El efecto de la misma queda ilustrado en la figura 3.5.

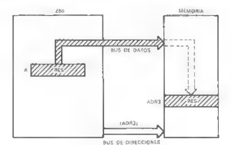


Figura 3.5
LD (ADR3), A (guardar el acumulador en la memoria).

Antes de la ejecución de la operación ADD, el acumulador contiene OP1 (véase la figura 3.4). Tras la suma se ha escrito en el mismo un nuevo resultado, que es "OP1 + OP2". Recuérde-

se que el contenido de cualquier registro interno del microprocesador, al igual que el de cualquier posición de memoria, permanece invariable tras una operación de lectura. En otras palabras, leer el contenido de un registro o de una posición de la memoria no altera ese contenido. Este cambia únicamente en las operaciones de *escritura*. En el caso que nos ocupa, los contenidos de las posiciones de memoria ADR1 y ADR2 permanecen invariables a lo largo de todo el programa. Sin embargo, tras la instrucción ADD, el contenido del acumulador sí se modifica, porque se ha escrito en él la salida de la ALU; en consecuencia, su contenido anterior se pierde.

En lugar de ADR1, ADR2 y ADR3 pueden utilizarse direcciones numéricas reales. Para trabajar con direcciones simbólicas es preciso utilizar lo que se llaman "seudoinstrucciones", que especifican el valor de tales direcciones simbólicas para que el programa ensamblador pueda reemplazarlas por las direcciones físicas verdaderas. Esas seudoinstrucciones podrían ser:

ADR1 = 100H
ADR2 = 120H
ADR3 = 200H

Ejercicio 3.1: *Consultando exclusivamente la tabla de instrucciones del final del libro, escríbase un programa que sume dos números almacenados en las posiciones de memoria LOC1 y LOC2 y deposite el resultado en la posición LOC3. Compárese el programa con el que acaba de estudiarse.*

SUMA DE 16 BITS

El programa de suma de 8 bits sirve únicamente para sumar números de 8 bits, es decir, números comprendidos entre 0 y 255, si se trabaja en notación binaria absoluta. En la práctica, casi siempre es preciso trabajar con números de 16 bits o más y, por tanto, recurrir a la *precisión múltiple*. Presentaremos aquí algunos ejemplos de operaciones aritméticas en 16 bits que fácilmente podrán extrapolarse a 24, 32 o más (siempre múltiplos de 8). Supondremos que el primer operando está almacenado en las posicions de memoria ADR1 y ADR1-1. Como OP1 es esta vez un número de 16 bits, necesitará dos posiciones de 8 bits. De la misma forma, OP2 se almacena en ADR2 y ADR2-1. El resultado se deposita en las direcciones ADR3 y ADR3-1. La figura 3.6 ilustra todo esto; H denota la mitad superior (bits 8 a 15) y L la inferior (bits 0 a 7).

Figura 3.6
Operandos de la sum
bits.

MEMORIA

ADR1 - 1	(OP1)H
ADR1	(OP1)L
ADR2-1	(OP2)H
ADR2	(OP2)L
ADR3 1	(RES)H
ADR3	(RES)L

Figura 3.6
Operandos de la suma de 16 bits

Este programa sigue la misma lógica general que el anterior. En primer lugar, se suman las dos mitades inferiores de los dos operandos, porque el microprocesador sólo puede operar de 8 en 8 bits. El acarreo que pudiera generar esa suma se almacena automáticamente en el bit interno de acarreo ("C"). A continuación se suman las dos mitades de orden superior de los dos operandos y el acarreo, y el resultado se guarda en la memoria. El programa es el siguiente:

```
LD   A, (ADR1)     CARGAR LA MITAD INFERIOR
                   DE OP1
LD   HL, ADR2      CARGAR DIRECCION DE LA MI-
                   TAD INFERIOR DE OP2
ADD  A, (HL)       SUMAR OP1 Y OP2 (INFERIORES)
LD   (ADR3), A     ALMACENAR EL RESULTADO (IN-
                   FERIOR)
LD   A, (ADR1-1)   CARGAR LA MITAD SUPERIOR
                   DE OP1
DEC  HL            DIRECCION DE LA MITAD SUPE-
                   RIOR DE OP2
ADC  A, (HL)       (OP1 + OP2) SUPERIOR + ACA-
                   RREO
LD   (ADR3-1), A   ALMACENAR EL RESULTADO
                   (SUPERIOR)
```

Las primeras cuatro instrucciones del programa son idénticas a las utilizadas para la suma de 8 bits de la sección anterior y dan lugar a la suma de las mitades menos significativas (bits 0 a 7) de OP1 y OP2. La suma, llamada "RES", se deposita en la posición de memoria ADR3 (véase la figura 3.6).

El posible acarreo ("0" ó "1") que pudiera resultar de la suma se almacena automáticamente en el bit de acarreo C del registro de estado (registro F). Si los dos números generan acarreo, el bit C será igual a "1"; en caso contrario, su valor será "0".

Las cuatro instrucciones siguientes son también básicamente iguales a las del programa de 8 bits a que nos estamos refiriendo. En este caso sirven para sumar las mitades más significativas (o mitades superiores, es decir, los bit 8 a 15) de OP1 y OP2 y cualquier acarreo, y almacenar el resultado en ADR3-1.

Tras la ejecución de este programa de 8 instrucciones, el resultado de 16 bits aparece almacenado en las posiciones de memoria ADR3 y ADR3-1. Se observará, no obstante, que hay una diferencia entre las dos partes del programa; en efecto, la *instrucción "ADD"* utilizada en la primera parte (instrucción tercera) no aparece en la segunda. Dicha instrucción suma dos operandos, con independencia del acarreo. En la segunda parte se ha empleado en su lugar otra, llamada "ADC", que suma dos operandos más cualquier acarreo que pudiera haberse producido y es, pues, necesaria para obtener un resultado correcto. Como la suma previamente ejecutada sobre las mitades inferiores puede dar lugar a un acarreo, es preciso tener éste en cuenta en las instrucciones de la segunda parte.

La cuestión que surge inmediatamente es: ¿qué ocurriría si la suma de las mitades superiores de los operandos también diese lugar a un acarreo? Hay dos posibilidades: la primera es suponer que se trata de un error: este programa está pensado para trabajar con resultados de hasta 16 bits, pero no de 17. La otra es incluir instrucciones adicionales para verificar precisamente la posibilidad de que se produzca un acarreo al final del programa. Se trata de la primera de una larga serie de decisiones que ha de tomar el programador.

Nota: Hasta ahora hemos supuesto que la mitad superior de un operando se almacena "encima" de la inferior, es decir, en la dirección de memoria inmediatamente inferior. Pero las cosas no son necesariamente así, y, de hecho, en el Z80 las direcciones se almacenan al revés: primero, la mitad inferior, y a continuación, la mitad superior en la siguiente posición de memoria. Con el fin de trabajar en una convención común para direcciones y datos, es recomendable que también éstos se almacenen

Figura 3.7
Almacenamiento de opera
en orden inverso.

con la parte inferior sobre la superior. La situación se ilustra en la figura 3.7.

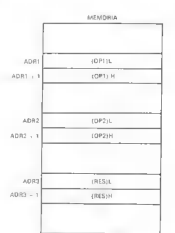


Figura 3.7
Almacenamiento de operandos en orden inverso.

Al trabajar con operandos de varios bytes, es importante tener en cuenta dos convenciones decisivas:

— el orden en que se almacenan los datos en memoria.
— la zona que señalan los apuntadores (byte inferior o byte superior).

Los ejercicios 3.2 y 3.3 están pensados para aclarar estas cuestiones.

Ejercicio 3.2: *Vuelva a escribir el programa de suma de 16 bits con la organización de memoria descrita en la figura 3.7.*

Ejercicio 3.3: *Supóngase ahora que ADR1 no señala hacia la mitad inferior de OP1 (como en las figuras 3.6 ó 3.7), sino hacia la superior (figura 3.8). Vuélvase a escribir el programa teniendo en cuenta esta nueva convención.*

Es el programador quien decide cómo se almacenan los números de 16 bits y también si las referencias de dirección señalan a la mitad inferior de dichos números o a la superior. Es otra decisión que hay que aprender a tomar durante el diseño de algoritmos y estructuras de datos.

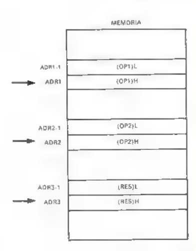


Figura 3.8
Punteros del byte superior.

Los programas que acabamos de examinar son programas tradicionales, que utilizan el acumulador. Veremos a continuación una alternativa al de 16 bits, que trabaja no con el acumulador, sino con algunas de las instrucciones especiales de 16 bits de que dispone el Z80. Supondremos que los operandos están almacenados tal como describe la figura 3.6. El programa es el siguiente:

```
LD   HL, (ADR1) CARGAR HL CON OP1
LD   BC, (ADR2) CARGAR BC CON OP2
ADD  HL, BC     SUMAR 16 BITS
LD   (ADR3), HL ALMACENAR RES EN ADR3
```

Lo primero que llama la atención en este programa es que es mucho más corto que el anterior. Se dice que es más "elegante". *Con ciertas limitaciones, los registros H y L del Z80 pueden utilizarse como un acumulador de 16 bits.*

Ejercicio 3.4: *Con las instrucciones de 16 bits que acabamos de presentar, escríbase un programa de suma para operandos de 32 bits, suponiendo que éstos se almacenan como describe la figura 3.9.*

Respuesta:

```
LD   HL, (ADR1)
LD   BC, (ADR2)
ADD  HL, BC
```

Figura 3.9
Suma de 32 bits.



examinar son programas
dor. Veremos a continua-
, que trabaja no con el
instrucciones especiales de
dremos que los operandos
la figura 3.6. El programa

L CON OP1
C CON OP2
BITS
AR RES EN ADR3

n en este programa es que
rior. Se dice que es más
os registros H y L del Z80
or de 16 bits.

le 16 bits que acabamos de
a de suma para operandos
almacenan como describe la

```
LD   (ADR3), HL
LD   HL, (ADR1 + 2)
LD   BC, (ADR2 + 2)
ADC  HL, BC
LD   (ADR3 + 2), HL
```

MEMORIA

ADR1 + 3	SUPERIOR
	OPR1
ADR1	INFERIOR
	SUPERIOR
	OPR2
ADR2	INFERIOR
	SUPERIOR
	RES
ADR3	INFERIOR

Figura 3.9
Suma de 32 bits.

Ahora que ya sabemos programar la suma binaria, podemos pasar a la resta.

RESTA DE NUMEROS DE 16 BITS

La resta de 8 bits es demasiado sencilla, así que la dejaremos como ejercicio y pasaremos directamente al problema de restar números de 16 bits. Como es habitual, los dos números OP1 y OP2 se almacenan en las direcciones ADR1 y ADR2, suponiendo una disposición de memoria similar a la ilustrada en la figura 3.7. Para restar se sustituye la instrucción ADD por la SBC.

Ejercicio 3.5: *Escribir un programa de resta.*

El programa aparece a continuación, y las rutas seguidas por los datos se muestran en la figura 3.10.

```
LD   HL, (ADR1)    OP1 EN HL
LD   DE, (ADR2)    OP2 EN DE
AND A              ELIMINAR ACARREO
SBC HL, DE         OP1 – OP2
LD   (ADR3), HL    RES EN ADR3
```

El programa es básicamente igual al de suma de 16 bits. Sin embargo, mientras que el Z80 tiene dos tipos de suma en registros dobles —ADD y ADC—, sólo cuenta con una resta: SBC. Como consecuencia de ello, ha habido que introducir dos cambios.



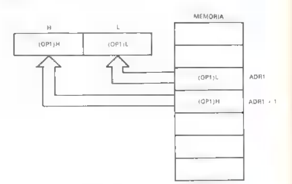


Figura 3.10
Carga de 16 bits. LD HL, (ADR1).

El primero es el uso de SBC en lugar de ADD.

El segundo es la instrucción "AND A", utilizada para eliminar la bandera de acarreo antes de restar. Esta instrucción no modifica el valor de A.

La precaución es necesaria, porque el Z80 dispone de dos formas de suma, con y sin acarreo, en los registros H y L, pero sólo de una resta, SBC, o resta con acarreo, cuando trabaja en el par de registro HL. Como SBC tiene en cuenta automáticamente el valor del bit de acarreo, es preciso poner éste a 0 antes de ejecutar la operación, y eso es precisamente lo que hace la instrucción "AND A".

Ejercicio 3.6: *Escríbase de nuevo el programa de resta sin usar las instrucciones especiales de 16 bits.*

Ejercicio 3.7: *Escríbase el programa de resta para operandos de 8 bits.*

Hay que recordar que, en la aritmética en complemento a dos, el valor final de la bandera de acarreo no tiene significado. Si, como resultado de la resta, se produce una situación de desbordamiento, entrará en juego el bit correspondiente (bit V) del registro de estado, cuyo valor podrá verificarse.

Los ejemplos que acabamos de estudiar corresponden a sencillas sumas y restas binarias, pero es necesario trabajar en otras formas de representación aritmética, y en particular en BDC.

Aritmética BCD

SUMA BCD DE 8 BITS

El concepto de notación aritmética BCD se expuso en el capítulo 1, pero recordaremos aquí sus características esenciales. Se utiliza, sobre todo, en aplicacions contables, en las que es de rigor conservar en el resultado todas las cifras significativas. En notación BCD se emplea un *nibble* de 4 bits para almacenar una cifra decimal (de 0 a 9), de manera que un byte de 8 bits representa dos cifras BCD *(BCD condensado)*. Veamos ahora cómo se suman dos bytes de dos cifras BCD cada uno.

Para identificar la naturaleza del problema, analizaremos antes algunos ejemplos numéricos.

Sea la suma de "01" y "02":

"01" se representa como	0000 0001
"02" se representa como	0000 0010
El resultado es	0000 0011

que es la representación BCD de "03" (si duda al deducir los equivalentes BCD, consulte la tabla de conversión del final del libro). Este caso ha sido muy fácil. Veamos otro:

"08" se representa como	0000 1000
"03" se representa como	0000 0011

Ejercicio 3.8: *Calcúlese la suma de los dos números de arriba en notación BCD. ¿Qué se obtiene como resultado?*

Respuesta: Si el resultado es "0000 1011", lo que ha obtenido es la suma *binaria* de 8 y 3, es decir, 11 en representación binaria. Lo que ocurre es que "1011" *no es un código BCD*

válido. De lo que se trata es de obtener la representación *BCD* de "11", es decir, 00010001.

El problema radica en que la representación BCD utiliza únicamente las primeras diez combinaciones de cuatro cifras para codificar los símbolos decimales 0 ■ 9. Las seis posibles combinaciones que quedan no se usan, y "1011" es una de ellas. En otras palabras, siempre que la suma de dos cifras BCD sea mayor que 9, hay que añadir 6 al resultado para saltar por encima los seis códigos que no se usan.

En efecto, al sumar la representación binaria de "6" a 1001

```
              1011    (resultado binario no permitido
            + 0110    (+ 6)
se obtiene:  00010001
```

que es "11" en notación BCD. Por fin hemos obtenido el resultado correcto.

Este ejemplo ilustra una de las dificultades básicas que plantea la notación BCD, a saber: la necesidad de compensar la existencia de seis códigos que no se usan. Para corregir el resultado de la suma binaria se utiliza una instrucción "DAA", llamada "ajuste decimal" (la instrucción suma 6 si el resultado es mayor que 9).

El mismo ejemplo servirá para ilustrar el problema siguiente. El acarreo procede de la cifra BCD inferior (la de la derecha) y pasa ■ la de la izquierda. Es preciso tener en cuenta este acarreo interno y sumarlo ■ la segunda cifra BCD, cosa que hace automáticamente la instrucción de suma. Sin embargo, con frecuencia conviene detectar este acarreo interno del bit 3 al 4 ("semiacarreo"), para lo que se emplea la bandera H.

Ilustraremos todo esto con el siguiente ejemplo, un programa que suma los números BCD "11" y "22":

```
LD   A, 11H      CARGAR EL LITERAL BCD "11"
ADD  A, 22H      SUMAR EL LITERAL BCD "22"
DAA              AJUSTE DECIMAL DEL RESULTA-
                 DO
LD   (ADR), A    ALMACENAR EL RESULTADO
```

En este programa hemos introducido un símbolo nuevo —"H"— que, situado dentro del campo del operando de la instrucción, indica que el dato al que sigue se expresa en notación hexadecimal. Las representaciones hexadecimal y BCD de las cifras "0" a "9" son idénticas. En este caso queremos sumar los literales (o constantes) "11" y "22", y almacenar el

Figura 3.11
Almacenamiento de cifras B

obtener la representación
1.

epresentación BCD utiliza
inaciones de cuatro cifras
es 0 a 9. Las seis posibles
n, y "1011" es una de ellas.
ma de dos cifras BCD sea
resultado para saltar por
usan.
ción binaria de "6" a 1001:

ado binario no permitido)

or fin hemos obtenido el

s dificultades básicas que
necesidad de compensar la
se usan. Para corregir el
za una instrucción "DAA".
ción suma 6 si el resultado

lustrar el problema siguien-
D inferior (la de la derecha)
eciso tener en cuenta este
unda cifra BCD, cosa que
de suma. Sin embargo, con
arreo interno del bit 3 al 4
plea la bandera H.
guiente ejemplo, un progra-
11" y "22":

EL LITERAL BCD "11"
LITERAL BCD "22"
ECIMAL DEL RESULTA-

AR EL RESULTADO

ducido un símbolo nuevo
campo del operando de la
que sigue se expresa en
aciones hexadecimal y BCD
as. En este caso queremos
11" y "22", y almacenar el

resultado en la dirección ADR. Cuando el operando se especifica como parte de una instrucción, como el ejemplo que nos ocupa, se habla de *direccionamiento inmediato* (en el capítulo 5 se analizarán en detalle las diversas formas de direccionamiento). Almacenar el resultado en una dirección especificada, como LD (ADR), A se llama *direccionamiento absoluto* cuando ADR representa una dirección de 16 bits.

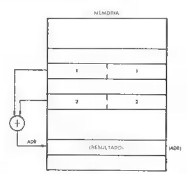


Figura 3.11
Almacenamiento de cifras BCD.

Este programa es análogo al de suma binaria de 8 bits, pero con la nueva instrucción "DAA", de la que mostraremos el funcionamiento con un ejemplo. Empecemos por sumar "11" y "22" en BCD:

```
  00010001  (11)
+ 00100010  (22)
= 00110011  (33)
    3   3
```

El resultado, alcanzado con las reglas de la suma binaria, es correcto.

Sumemos ahora "22" y "39" con las mismas reglas de la suma *binaria:*

```
  00100010  (22)
+ 00111001  (39)
= 01011011
    5   ?
```

"1011" es un *código BCD no permitido*, porque en BCD se utilizan sólo los primeros diez códigos binarios y se saltan los

seis siguientes; por tanto, habrá que hacer aquí lo mismo, es decir, sumar 6 al resultado:

```
    01011011  (resultado binario)
+       0110  (6)
= 01100001  (61)
   6    1
```

que es el resultado BCD correcto.

Ejercicio 3.9: *¿Podría colocarse en el programa la instrucción DAA después de la LD (ADR), A?*

RESTA BCD

A primera vista, la resta en BCD parece una cosa muy complicada. La operación se realiza sumando el *complemento a diez* del número, igual que se sumaba el complemento a dos para realizar la resta binaria. El complemento a 10 se calcula determinando el de 9 y sumando 1, lo que en un microprocesador normal consume de tres a cuatro operaciones; sin embargo, el Z80 dispone de una potente instrucción DAA, que simplifica las cosas.

La instrucción DAA ajusta automáticamente el valor del resultado del acumulador en función del valor de las banderas C, H y N, antes de DAA, a su valor correcto (véase el capítulo siguiente para más detalles sobre DAA).

SUMA BCD DE 16 BITS

Esta operación se realiza exactamente igual que la binaria correspondiente. El programa es el siguiente:

```
LD    A,(ADR1)       CARGAR (OP1) EN A
LD    HL,(ADR2)      CARGAR ADR2 EN HL
ADD   A,(HL)         (OP1 + OP2) INFERIOR
DAA                  AJUSTE DECIMAL
LD    (ADR3),A       ALMACENAR EL RESULTADO (IN-
                     FERIOR)
LD    A,(ADR1 + 1)   CARGAR (OP1) H EN A
INC   HL             PUNTERO A ADR2 + 1
ADC   A,(HL)         (OP1 + OP2)  SUPERIOR + ACA-
                     RREO
```

```
DAA                  AJUSTE DECIMAL
LD     (ADR3 + 1), A ALMACENAR   EL   RESULTADO
                     (SUPERIOR)
```

RESTA EN BCD EMPAQUETADO

Ya se han descrito los procedimientos elementales de suma y resta BCD. Sin embargo, en la práctica habitual, los números BCD tienen un número variable de bytes. Como ejemplo simplificado de resta en BCD empaquetado supondremos que las dos cantidades localizadas en N1 y N2 tienen el mismo número de bytes BCD, número que se llama CUENTA. La organización de registros y memoria se muestra en la figura 3.12. El programa es el siguiente:

```
BCDPAK LD     B, CUENTA
       LD     DE, N2
       LD     HL, N1
       AND    A          ELIMINAR ACARREO
MENOS  LD     A, (DE)    BYTE N2
       SBC    A, (HL)    N2 - N1
       DAA
       LD     (HL), A    ALMACENAR EL RE-
                         SULTADO
       INC    DE
       INC    HL
       DJNZ   MENOS      DECREMENTAR B Y
                         REPETIR HASTA B = 0
```

N1 y N2 son las direcciones en que están almacenados los números BCD, direcciones que se cargan en los pares de registros DE y HL:

```
BCDPAK LD     B, CUENTA
       LD     DE, N2
       LD     HL, N1
```

A continuación, antes de la primera resta, hay que eliminar el bit de acarreo. Ya se ha dicho que hay varias formas de hacerlo. Una de ellas es:

```
AND A
```

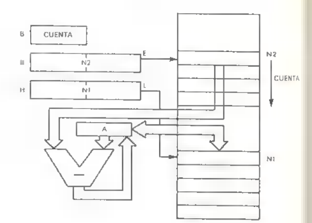


Figura 3.12
Resta en BCD empaquetado
N1 ← N2 − N1.

El primer byte de N2 se carga en el acumulador y se le resta el primero de N1. A continuación se utiliza la instrucción DAA para obtener el valor BCD correcto:

```
MENOS  LD    A, (DE)
       SBC   A, (HL)
       DAA
```

El resultado se almacena en N1:

```
LD   (HL), A
```

Por último, se incrementan los punteros de los bytes en curso:

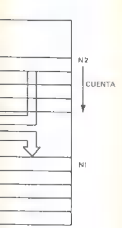
```
INC  DE
INC  HL
```

Se decrementa el contador y se ejecuta el bucle de resta hasta que alcance el valor "0":

```
DJNZ  MENOS
```

La instrucción DJNZ es una instrucción especial del Z80 que decrementa el registro B y salta —si no es 0— en una sola operación.

Multiplicació



el acumulador y se le resta
utiliza la instrucción DAA
o:

:

punteros de los bytes en

cuta el bucle de resta hasta

ucción especial del Z80 que
si no es 0— en una sola

Ejercicio 3.10: *Compárese el programa que acaba de presentarse con el de suma binaria de 16 bits. ¿Dónde está la diferencia?*

Ejercicio 3.11: *¿Son intercambiables los papeles de DE y HL? (Un consejo: cuidado con SBC.)*

Ejercicio 3.12: *Escríbase un programa de resta en BCD de 16 bits.*

BANDERAS BCD

En notación BCD, la bandera de acarreo que aparece como resultado de una suma indica que el resultado es superior a 99. La situación es diferente a la que se daba en complemento a dos, porque las cifras BCD están representadas en auténtica notación binaria; por el contrario, la presencia de bandera de acarreo tras una resta indica un defecto.

TIPOS DE INSTRUCCIONES

Hemos utilizado ya dos tipos de instrucciones del microprocesador: instrucciones LD, que cargan el acumulador a partir de direcciones de memoria o almacenan su contenido en direcciones especificadas; se llaman instrucciones de *transferencia de datos*.

Instrucciones *aritméticas*, como ADD, SUB, ADC y SBC, que ejecutan operaciones de suma y resta. Pronto veremos en este mismo capítulo otras instrucciones de la ALU.

Pero hay otros tipos que todavía no hemos usado: se trata de las instrucciones de *salto*, que modifican el orden de ejecución del programa; recurriremos a esta clase de instrucciones en el próximo ejemplo. Conviene observar que a las instrucciones de *salto* se les llama también de *bifurcación* en situaciones condicionales, es decir, en puntos del programa en los que se toma una decisión lógica. Como sugiere el nombre de *bifurcación*, la presencia de esa instrucción escinde el camino único del programa en dos divergentes.

Multiplicación

Analicemos a continuación un problema aritmético más complicado: la multiplicación de números binarios. Antes de redactar el algoritmo de la operación, examinaremos una multiplicación decimal corriente; sea la de 12 por 23:

12	(multiplicando = MPD)
× 23	(multiplicador = MPR)
36	(producto parcial)
+ 24	
= 276	(resultado final)

La operación se lleva a cabo multiplicando la cifra de la derecha del multiplicador por el multiplicando, es decir: "3" × "12"; el producto parcial es "36"; a continuación se multiplica la siguiente cifra del multiplicador, "2", por "12", y el resultado, "24", se suma al producto parcial.

Pero todavía falta una operación: 24 se escribe *desplazado una posición hacia la izquierda* (decimos que 24 se *desplaza a la izquierda* una posición, pero igual podríamos haber dicho que el producto parcial 36 se *desplaza una posición a la derecha*).

Los dos números, correctamente desplazados, se suman, y así se obtiene el producto 276. Es algo muy sencillo, y las cosas ocurren exactamente de la misma forma en notación binaria.

Veamos, por ejemplo, la multiplicación de 5 por 3:

(5)	101	(multiplicando
(3)	× 011	(multiplicador)
	101	(producto parcial)
	101	
	000	
(15)	01111	(resultado final)

Figura 3.13
Diagrama de flujo del alg
básico de multiplicación

Realizaremos la multiplicación exactamente igual que en el ejemplo que acabamos de ver. La representación formal del algoritmo aparece en la figura 3.13. Se trata de un diagrama de flujo —el primero del libro—, y lo analizaremos con cierto detalle.

El diagrama de flujo es una representación simbólica del algoritmo que acabamos de seguir. Cada rectángulo representa una orden que debe ejecutarse, y que habrá que transformar en una o más instrucciones del programa. En cada uno de los símbolos romboidales hay que llevar a cabo una comprobación, ya que son *puntos de bifurcación* del programa. Si el resultado de la comprobación es afirmativo, la bifurcación conduce el flujo del programa a una posición determinada; en caso negativo, lo conduce a una posición diferente. La idea de bifurcación se expondrá más adelante, en el propio programa. El lector deberá, por el momento, estudiar atentamente el diagrama de flujo hasta que adquiera la completa seguridad de que represen-

ta efectivamente el algoritmo de multiplicación. Obsérvese que del rombo inferior parte una flecha que se dirige al superior; se debe a que esa porción del diagrama debe ejecutarse ocho veces, una por cada bit del multiplicador. Esta disposición del programa que provoca el reinicio de una misma operación en un mismo punto es lo que se llama un *bucle*.

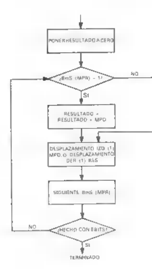


Figura 3.13
Diagrama de flujo del algoritmo básico de multiplicación.

Ejercicio 3.13: *Ejecútese la multiplicación binaria de "4" por "7" utilizando el diagrama de flujo y verificando el resultado, que debe ser "28". Pruébese de nuevo si el que se obtiene es otro, porque sólo quien sea capaz de seguir el diagrama hasta dar con el valor correcto estará en condiciones de transformarlo en un programa.*

MULTIPLICACION 8 POR 8

Vamos, por fin, a transformar el diagrama de flujo en un programa para el Z80, programa que aparece completo en la figura 3.14 y que estudiaremos en detalle. Como se recordará del capítulo 1, programar consiste en este caso en "traducir" el diagrama de flujo de la figura 3.13 al programa de la 3.14. Cada

uno de los recuadros del diagrama dará lugar a una o más instrucciones.

Se supone que MPR y MPD han recibido ya un valor concreto.

```
MPY88  LD   BC,(MPRAD)   CARGAR MULTIPLI-
                         CADOR EN C
       LD   B,8          B ES EL CONTADOR
                         DE BIT
       LD   DE,(MPDAD)   CARGAR EL MULTI-
                         PLICANDO EN E
       LD   D,0          BORRAR D
       LD   HL,0         PONER EL RESULTA-
                         DO A 0
MULT   SRL  C            DESPLAZAR EL BIT
                         DEL MULTIPLICA-
                         DOR AL ACARREO
       JR   NC,NOADD     VERIFICAR EL
                         ACARREO
       ADD  HL,DE        SUMAR MPD AL RE-
                         SULTADO
NOADD  SLA  E            DESPLAZAR MPD A
                         LA IZQUIERDA
       RL   D            LLEVAR BIT A D
       DEC  B            DECREMENTAR
                         CONTADOR DE DES-
                         PLAZAMIENTO
       JP   NZ,MULT      REPETIR SI
                         CONTADOR ≠ 0
       LD   (RESAD),HL   ALMACENAR EL RE-
                         SULTADO
```

Figura 3.14
Programa de multiplicación 8 × 8.

Figura 3.15
Registros usados en la mul[tipli]cación 8 × 8.

La primera casilla del diagrama es la de *inicialización*, necesaria porque antes de nada hay que poner a 0 una serie de registros o posiciones de memoria para que el programa trabaje en ellos. Los registros que se utilizarán en el programa de multiplicación aparecen en la figura 3.15.

Se utilizan tres pares de registros del Z80. El multiplicador de 8 bits se supone que reside en la posición de memoria MPRAD; el multiplicando, MPD, ocupa la dirección MPDAD, y los dos se cargarán en los registros C y E, respectivamente (véase la figura 3.15). El registro B trabaja como contador.

Los registros D y E albergan el multiplicando a medida que se desplaza de bit en bit hacia la izquierda.

dará lugar a una o más

han recibido ya un valor

CARGAR MULTIPLICADOR EN C
B ES EL CONTADOR DE BIT
CARGAR EL MULTIPLICANDO EN E
BORRAR D
PONER EL RESULTADO A 0
DESPLAZAR EL BIT DEL MULTIPLICADOR AL ACARREO
VERIFICAR EL ACARREO
SUMAR MPD AL RESULTADO
DESPLAZAR MPD A LA IZQUIERDA
LLEVAR BIT A D
DECREMENTAR CONTADOR DE DESPLAZAMIENTO
REPETIR SI CONTADOR ≠ 0
ALMACENAR EL RESULTADO

la de *inicialización*, necesa-
er a 0 una serie de registros
programa trabaje en ellos.
programa de multiplicación

del Z80. El multiplicador
la posición de memoria,
upa la dirección MPDAD,
os C y E, respectivamente
trabaja como contador.
ultiplicando a medida que
zquierda.

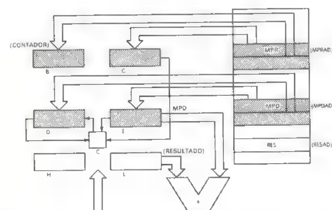


Figura 3.15
Registros usados en la multiplicación 8 × 8.

Obsérvese que, aunque en un principio basta con cargar C y E, hay que prever 16 bits para que también puedan cargarse B y D a partir de la memoria; se ponen, respectivamente, a "8" y "0".

Por último, hay que tener en cuenta que el resultado de una multiplicación de 8 por 8 bits puede ocupar hasta 16 bits, porque $2^8 \times 2^8 = 2^{16}$; por tanto, hay que reservar para el resultado dos registros, que son los H y L, como indica la figura 3.15.

El primer paso es cargar los registros B, C y E con los contenidos adecuados e iniciar el resultado (el producto parcial) al valor "0", tal como especifica el diagrama de flujo de la figura 3.13. Todo ello se consigue con las siguientes instrucciones:

```
MPY88  LD  BC,(MPRAD)
       LD  B,8
       LD  DE,(MPDAD)
       LD  D,0
       LD  HL,0
```

Las tres primeras instrucciones cargan MPR en el par de registros BC, el valor "8" en el registro B y MPD en el par de registros DE, respectivamente. Como MPR y MPD son palabras de 8 bits, se cargan, de hecho, en los registros C y E, mientras que las palabras de la memoria que les siguen pasan a B y D. La situación se ilustra en las figuras 3.16 y 3.17. La siguiente instrucción pone a 0 el contenido de D.

En este programa de multiplicación, el multiplicando se desplaza hacia la izquierda antes de sumarlo al resultado (recuérdese que también se puede desplazar el resultado a la derecha, como indica la cuarta casilla del diagrama de flujo de la figura 3.13). A cada paso, el multiplicando MPD se desplaza hacia el registro D, que, por tanto, debe iniciarse al valor "0", operación que lleva a cabo la instrucción cuarta. La quinta pone a 0 de una vez los contenidos de los registros H y L.

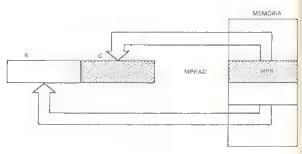


Figura 3.16
LD BC, (MPRAD).

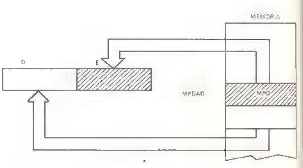
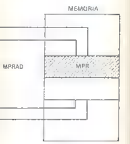


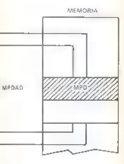
Figura 3.17
LD DE, (MPDAD).

El siguiente paso del diagrama de flujo consiste en poner a prueba el bit menos significativo (el de la derecha) del multiplicador, MPR. Si resulta ser "1", el valor de MPD se añade al resultado parcial; en caso contrario, no se añade. Para ello hacen falta tres instrucciones:

```
MULT  SRL   C
      JR    NC, NOADD
      ADD   HL, DE
```

ción, el multiplicando se
de sumarlo al resultado
esplazar el resultado a la
a del diagrama de flujo de
licando MPD se desplaza
debe iniciarse al valor "0".
rucción cuarta. La quinta
s de los registros H y L.





e flujo consiste en poner a
le la derecha) del multipli-
alor de MPD se añade al
, no se añade. Para ello

El primer problema a resolver es la verificación del bit menos significativo del multiplicador contenido en el registro C. Podemos usar para ello la instrucción BIT del Z80, que permite comprobar cualquier bit de cualquier registro, pero en este caso lo que nos interesa es crear un programa con un bucle lo más sencillo posible. Para utilizar la instrucción BIT, tendríamos que comprobar, primero, el bit 0; luego, el 1, y así sucesivamente hasta llegar al 7, lo que exigiría una instrucción diferente cada vez, algo incompatible con un solo bucle. Para acortar el programa hay que buscar otro camino, y en este caso hemos decidido trabajar con una instrucción de *desplazamiento*.

Nota: Hay una forma de utilizar la instrucción BIT y un bucle, pero exigiría que el programa se modificase a sí mismo, una práctica que, por el momento, evitaremos.

SRL es un nuevo tipo de operación que se ejecuta dentro de la unidad aritmética y lógica. Significa *desplazamiento lógico a la derecha*. Tras un desplazamiento lógico a la derecha, aparece un "0" en la posición del bit 7; por el contrario, tras un *desplazamiento aritmético a la derecha*, el bit que ocupa la posición 7 adopta el mismo valor que antes tenía dicha posición. En el próximo capítulo se describirán las diversas operaciones de desplazamiento. El resultado de la instrucción SRL C viene mostrado en la figura 3.15 por una flecha que sale del registro C y se dirige hacia el cuadrado que designa el bit de acarreo ("C"). En este punto, el bit de la derecha del MPR estará en el bit de acarreo C, el que se comprueba.

La instrucción siguiente, "JR NC, NOADD", es una operación de *salto* que significa: "si no hay acarreo" (NC), saltar a la dirección NOADD (etiqueta). Si el contenido del bit de acarreo es "0" (no hay acarreo), el programa salta a la dirección NOADD; si el contenido de C es "1" (hay acarreo), no se produce bifurcación, y se ejecuta la siguiente instrucción de la secuencia, en este caso "ADD HL, DE".

Esta instrucción dice que hay que sumar los contenidos de D y E a los de H y L, y dejar el resultado en H y L. Como E contiene el multiplicando, MPD (véase la figura 3.15), resulta que la instrucción suma dicho multiplicando al resultado parcial.

En este punto, con independencia de que MPD se haya sumado o no al resultado, hay que desplazar el multiplicando a la izquierda (cuarto recuadro del diagrama de flujo de la figura 3.13). Para ello se usa la instrucción:

NOADD SLA E

SLA significa "desplazamiento aritmético a la izquierda". Como ya hemos explicado, hay dos tipos de operaciones de desplaza-

miento: lógico y aritmético. Este es el aritmético. En el caso de desplazamiento a la izquierda. SLA especifica que el bit de la parte derecha del registro (el menos significativo) sea "0", como en el caso de SRL que vimos antes.

Supongamos, por ejemplo, que el contenido inicial del registro E fuera 00001001. Tras la instrucción SLA, ese contenido será 00010010, y el bit de acarreo valdrá 0.

Pero, como se ve en la figura 3.15, lo que nos interesa es desplazar el bit más significativo (BMS) de E directamente a D (este movimiento viene ilustrado por la flecha que va de E a D); sin embargo, no hay ninguna instrucción para desplazar un doble registro, como el D y E, de una vez. Una vez desplazados los contenidos de D y E, el bit de la izquierda habrá "caído" en el bit de acarreo; por tanto, hay que recoger ese bit y desplazarlo al registro D, y para ello sirve la instrucción siguiente:

RL D

RL es también una operación de desplazamiento, pero de distinto tipo. Significa "rotación circular a la izquierda". En una operación de *rotación circular*, al contrario que en una de *desplazamiento*, el bit que llega al registro es el contenido del bit de acarreo C (véase la figura 3.18), que es justamente lo que nos interesa. El contenido de C se carga en el extremo derecho de D, lo que, de hecho, equivale a transferir el bit izquierdo de E.

Esta secuencia de dos instrucciones se muestra en la figura 3.19. Como se ve, el bit identificado por X en la posición más significativa de E pasa primero al bit de acarreo y a continuación a la posición menos significativa de D.

Figura 3.18
Desplazamiento y rotación circular.

Figura 3.19
Desplazamiento de E a D

En este punto, como indica el diagrama de flujo de la figura 3.13, hay que señalar el siguiente bit de MPR y comprobar si es el octavo. Esto se consigue decrementando el contador de bits del registro B (figura 3.15). De decrementar el registro, se encarga la instrucción:

DEC B

que es una instrucción de decremento de resultado evidente.

Por último, es preciso comprobar si el contador ha disminuido hasta 0, lo que se consigue examinado el valor del bit Z. Como recordará el lector, la bandera Z (0) indica si la operación aritmética previa (DEC, por ejemplo) ha producido un resultado nulo. Obsérvese, sin embargo, que DEC HL, DEC BC, DEC DE, DEC IX y DEC SP no afectan a la bandera Z mencionada. Si el contador no es "0", quiere decir que la operación no ha terminado, y que hay que ejecutar una vez más

el bucle del programa, de lo que se encarga la instrucción siguiente:

JP NZ, MULT

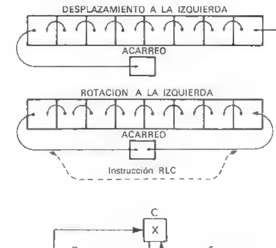


Figura 3.18
Desplazamiento y rotación circular.

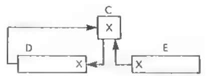


Figura 3.19
Desplazamiento de E a D.

Se trata de una instrucción de salto, la cual especifica que siempre que el bit Z no sea 0 (NZ significa no cero), hay que saltar a la posición MULT. De esta forma se cierra el *bucle del programa*, que se ejecutará una y otra vez hasta que el valor de B se reduzca a 0. En ese momento, el bit Z adquirirá un valor no nulo, y la instrucción JP NZ dejará de actuar, lo que dará lugar a que se ejecute la instrucción siguiente de la secuencia, a saber:

LD (RESAD), HL

Esta instrucción almacena el contenido de H y L, es decir, el resultado de la multiplicación, en la dirección RESAD. Obsérvese que la instrucción transfiere los contenidos de ambos registros a dos posiciones de memoria consecutivas: RESAD y RESAD + 1. Almacena 16 bits de una vez.

Ejercicio 3.14: *¿Sería capaz de escribir el programa de multiplicación que acabamos de estudiar sustituyendo la instrucción SRL C por la BIT (descrita en el capítulo siguiente)? ¿Qué inconveniente tiene?*

Tratemos ahora de mejorar el programa, si tal cosa es posible.

Ejercicio 3.15: *¿Puede sustituirse JR por JP al final del programa? En caso afirmativo, ¿qué ventaja tiene el cambio?*

Ejercicio 3.16: *¿Puede utilizarse DJNZ para acortar el final del programa?*

Ejercicio 3.17: *Estúdiense las instrucciones LD D,0 y LD HL,0 del principio del programa; ¿pueden sustituirse por*

```
XOR  A
LD   D,A
LD   H,A
LD   L,A
```

En caso afirmativo, ¿qué efecto ejercerían sobre el tamaño (número de bytes) y la velocidad?

En la mayor parte de los casos, el programa que acabamos de desarrollar será un subrutina que tendrá como instrucción final RET (*return*, vuelta). El mecanismo de subrutina se explicará más adelante en este mismo capítulo.

UN EJERCICIO IMPORTANTE

El que acabamos de ver es el primer programa realmente importante del libro. Incluye instrucciones muy diversas: de transferencia (LD), aritméticas (ADD), lógicas (SRL, SLA, RL) y de salto (JR, JP) y cuenta, además, con un bucle de siete instrucciones que empiezan en la dirección MULT y se ejecutan varias veces seguidas. Para aprender a programar es imprescindible entender perfectamente este programa. Es más largo que los sencillos programas aritméticos de los capítulos anteriores, y resulta imprescindible examinarlo con detenimiento. A continuación se propone un ejercicio de la mayor importancia, y es decisivo que el lector lo realice por completo y correctamente antes de seguir, porque será la única demostración de que ha asimilado los conceptos expuestos con anterioridad. Quien lo resuelva correctamente tendrá la seguridad de haber comprendi-

Figura 3.20
Tabla del ejercicio de mul cación.

do cabalmente la forma en que el microprocesador manipula la información, la desplaza entre los registros y la memoria, y la procesa. Quien no haga el ejercicio o quien no lo resuelva correctamente es muy probable que tropiece con dificultades al escribir sus propios programas. Aprender a programar exige práctica; por tanto, tome un papel, o utilice la figura 3.20, y haga el

Ejercicio 3.18: *Siempre que se escribe un programa es preciso comprobarlo a mano, para tener la seguridad de que proporciona resultados correctos, y eso es justamente lo que vamos a hacer ahora: la finalidad de este ejercicio es rellenar la tabla de la figura 3.20.*

ETIQUETA	INSTRUCCION	B	C	C (ACARREO)	D	E	H	L

Figura 3.20
Tabla del ejercicio de multiplicación.

La respuesta puede escribirse directamente en la figura o en un papel aparte. De lo que se trata es de indicar el contenido de cada uno de los registros que intervienen en el programa tras la ejecución de cada una de las instrucciones. En la figura 3.20

aparecen todos los que forman parte del programa de la figura 3.14, que son, de izquierda a derecha: B y C, el acarreo C, D y E, y H y L. En la parte izquierda de la figura se anotan la etiqueta, si existe, y la instrucción que va a ejecutarse. En la parte derecha se anota el contenido de cada uno de los registros tras la ejecución de la instrucción que figura a la izquierda. Si el contenido de un registro no se conoce (es indefinido), se señala tal circunstancia con un trazo. Como ejemplo, empezaremos a rellenar las primeras filas de la tabla:

ETIQUETA	INSTRUCCION	B	C	C	D	E	H	L
		--	--	-	--	--	--	--
MPY88	LD BC,(0200)	00	03	-	--	--	--	--

Figura 3.21
El programa de multiplicación tras una instrucción.

Suponemos en este caso que estamos multiplicando "3" (MPR) por "5" (MPD).

La primera instrucción que se ejecuta es "LD BC, (MPRAD)". El contenido de la posición de memoria MPRAD se carga en los registros B y C. Hemos supuesto que MPR es igual a 3, es decir, "00000011". Tras la ejecución de esta instrucción, el contenido del registro C pasa a ser "3". Obsérvese que la instrucción también carga B con lo que siga a MPR en la memoria. La instrucción siguiente se ocupa de ello y carga en B el valor "8", como ilustra la figura 3.22. Por el momento los contenidos de D y E y H y L están indefinidos. La instrucción LD no perturba al bit de acarreo, de tal manera que también está indefinido el contenido del mismo.

ETIQUETA	INSTRUCCION	B	C	C	D	E	H	L
		--	--	-	--	--	--	--
MPY88	LD BC,(0200)	00	[illegible]	-	--	--	--	--
	LD B, 08	08	03	-	--	--	--	--

Figura 3.22
El programa de multiplicación tras dos instrucciones.

La situación tras la ejecución de las primeras cinco instrucciones del programa (justo antes de MULT) se ilustra en la figura 3.23.

La instrucción SRL lleva a cabo un desplazamiento lógico a la derecha, de manera que el bit de ese extremo de MPR pasa al bit de acarreo. Como se ve en la figura 3.24, el contenido de MPR tras el desplazamiento es "0000 0001". El bit de acarreo C vale ahora "1". La operación no ha afectado a los demás

Figura 3.23
■ programa de multiplic
tras cinco instrucciones.

Figura 3.24
Un pase del bucle.

registros. Ahora, debe continuar usted mismo rellenando la tabla.

Al final de este capítulo, en la figura 3.42, se recoge una segunda repetición.

Figura 3.23
El programa de multiplicación tras cinco instrucciones.

ETIQUETA	INSTRUCCION	B	C	C	D	E	H	L
		--	--	-	--	--	--	--
MPY88	LD BC,(0200)	00	03	-	--	--	--	--
	LD B, 08	08	03	-	--	--	--	--
	LD DE,(0202)	08	03	-	00	05	--	--
	LD D, 00	08	03	-	00	05	--	--
	LD HL,0000	08	03	-	00	05	00	00

Figura 3.24
Un paso del bucle.

ETIQUETA	INSTRUCCION	B	C	C	D	E	H	L
		--	--	-	--	--	--	--
MPY88	LD BC,(0200)	00	03	-	--	--	--	--
	LD B, 08	08	03	-	--	--	--	--
	LD DE,(0202)	08	03	-	00	05	--	--
	LD D, 00	08	03	-	00	05	--	--
	LD HL,0000	08	03	-	00	05	00	00
MULT	SRL C	08	01	1	00	05	00	00
	JR NC,0114	08	01	1	00	05	00	00
	ADD HL,DE	08	01	0	00	05	00	05
NOADD	SLA E	08	01	0	00	0A	00	05
	RL D	08	01	[illegible]	00	0A	00	05
	DEC B	07	01	0	00	0A	00	05
	JP NZ,010F	07	01	0	00	0A	00	05

La figura 3.40 recoge el listado completo de los contenidos de todos los registros y banderas del Z80. En la figura 3.41 aparece un listado decimal y hexadecimal.

OTRAS POSIBILIDADES DE PROGRAMACION

El programa que acabamos de desarrollar puede escribirse de forma distinta. Como norma general, el programador debe estar en condiciones de modificar y mejorar cualquier progra-

ma. En este caso se ha desplazado el multiplicando antes de sumar, pero matemáticamente hubiera sido lo mismo desplazar el resultado una posición a la derecha antes de sumarlo al multiplicando. De hecho, se trata de un ejercicio interesante.

Ejercicio 3.19: *Escríbase un programa de multiplicación de 8 × 8 con el mismo algoritmo ya visto, pero desplazando el resultado una posición a la derecha, en lugar de hacer lo mismo hacia la izquierda con el multiplicando. Compárese con el programa anterior, y determínese si el nuevo tratamiento será o no más rápido. Las velocidades de ejecución de las instrucciones del Z80 se recogen en el capítulo siguiente.*

Figura 3.25
Programa de multiplicación mejorado, paso 1.

PROGRAMA DE MULTIPLICACION MEJORADO

El programa que acabamos de estudiar es una traducción directa del algoritmo. Sin embargo, *para programar con eficacia hay que dedicar atención al detalle*, para ver si puede acortarse el programa o acelerarse su velocidad de ejecución. Veamos ahora algunas opciones que mejoran el programa de multiplicación.

Paso 1

Una posibilidad de mejora consiste en aprovechar mejor las instrucciones del Z80. Así, las instrucciones penúltima y antepenúltima pueden reemplazarse por una sola:

```
DJNZ MULT
```

Se trata de una instrucción especial del Z80 de "salto automático" que decrementa el registro B y bifurca a una posición determinada si no vale "0". Hablando estrictamente, la instrucción no equivale por completo a las otras dos:

```
DEC B
JP NZ, MULT
```

porque especifica un *desplazamiento*, y sólo puede darse un salto dentro del intervalo − 126 a + 129. Sin embargo, en este caso debemos saltar a una posición alejada tan sólo unos pocos bytes, por lo que la mejora es factible. El programa resultante aparece en la figura 3.25.

Figura 3.25
Programa de multiplicación mejorado, paso 1.

```
MPY88B  LD     DE,(MPDAD)
        LD     BC,(MPRAD)
        LD     B,8          CONTADOR DE
                            BIT
        LD     HL,0
MULT    SRL    C
        JR     NC,NOADD
        ADD    HL,DE
NOADD   SLA    E
        RL     D
        DJNZ   MULT
        LD     (RESAD),HL
        RET
```

Paso 2

Como puede observarse en el programa inicial de la figura 3.14, se emplean tres operaciones de desplazamiento diferentes: el multiplicador se desplaza a la derecha, a continuación se desplaza a la izquierda el multiplicando MPD en dos operaciones, un desplazamiento a la izquierda del registro E más una permutación circular a la izquierda del D. Todo esto consume tiempo. Hay un "truco" habitual en la programación de la multiplicación, que se basa en el hecho de que cada vez que el multiplicador se desplaza un bit a la derecha, en el registro multiplicador queda libre otro bit. Así, suponiendo que el desplazamiento se produce hacia la derecha —caso del ejemplo anterior— queda libre un bit a la izquierda. Se observa también que el primer producto parcial (o "resultado") utiliza, como máximo, 9 bits. Si al principio hubiésemos reservado para el resultado un solo registro, podríamos haber utilizado la posición que deja libre el multiplicador para almacenar el noveno bit de aquél.

Tras el siguiente desplazamiento de MPR, el tamaño del producto parcial vuelve a aumentar en un bit, de tal manera que puede reservarse al principio un solo registro para el producto parcial y utilizar la posición libre que se produce al desplazar MPR; por tanto, para mejorar el programa, asignaremos MPR y RES a un par de registros. Lo ideal sería desplazarlos juntos en una sola operación, pero el Z80 sólo es capaz de desplazar 8 bits de una vez; como la mayor parte de los microprocesadores de 8 bits, no dispone de las instrucciones necesarias para desplazar 16 bits. (ADD HL,HL desplazan los 16 bits de HL una posición a la izquierda.)

Pero queda otro recurso. El Z80 —como el 8080— dispone de instrucciones especiales de adición de 16 bits que ya hemos

utilizado en este libro. Si el multiplicador y el resultado están almacenados en los registros apareados H y L, podemos usar la instrucción,

ADD HL,HL

que suma los contenidos de H y L a sí mismos. Sumar un número a sí mismo equivale a duplicarlo, y en el sistema binario, duplicar un número equivale a desplazarlo hacia la izquierda una posición; por tanto, hemos realizado un desplazamiento de 16 bits de una sola vez. Por desgracia, el desplazamiento se ha producido hacia la izquierda y no hacia la derecha, como queríamos, pero no hay problema.

Conceptualmente, MPR puede desplazarse tanto a la izquierda como a la derecha. Hemos utilizado el algoritmo de desplazamiento a la derecha, porque es el que se usa en la suma convencional, pero no hay necesidad de hacer así las cosas. La operación de suma es conmutativa y admite la inversión del orden, por lo que da lo mismo desplazar MPR a la izquierda.

Para sacar partido a este desplazamiento simulado de 16 bits tendremos que desplazar MPR a la izquierda, por lo que éste residirá en el registro H, y el resultado en el L. La configuración de registros resultante se ilustra en la figura 3.26.

Figura 3.27
Programa mejorado de m
cación, paso 2.

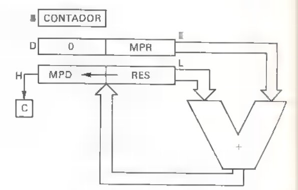


Figura 3.26
Registros del programa mejorado de multiplicación.

El resto del programa es básicamente igual al anterior. Aparece en la figura 3.27.

Al comparar este programa con el anterior se observa que se ha reducido la longitud del bucle de multiplicación (el número de instrucciones entre MULT y el salto). Este progra-

ma tiene menos instrucciones, y, en principio, avanzará con más rapidez, lo que demuestra la importancia de almacenar la información en los registros idóneos.

```
MUL88C  LD    HL,(MPRAD-1)
        LD    L,0
        LD    DE,(MPDAD)
        LD    D,0
        LD    B,8            CONTADOR
MULT    ADD   HL, HL         DESPLAZA-
                             MIENTO A LA
                             IZQUIERDA
        JR    NC,NOADD
        ADD   HL,DE
NOADD   DJNZ  MULT
        LD    (RESAD),HL
        RET
```

Figura 3.27
Programa mejorado de multiplicación, paso 2.

El diseño de programas "directo" da, por lo general, resultados que funcionan, pero que no son *óptimos*. Es, pues, importante aprender a sacar el máximo partido a los registros e instrucciones disponibles. Los ejemplos que hemos visto constituyen un enfoque racional de la selección de registros e instrucciones con vistas a optimizar la eficacia.

Ejercicio 3.20: *Calcúlese la velocidad de multiplicación con el último programa. Supóngase que tiene lugar una bifurcación en el 50 por 100 de los casos. El número de ciclos consumidos por cada una de las instrucciones aparece en el capítulo siguiente; la frecuencia del reloj será de 2 MHz (un ciclo = 0,5 μs).*

Ejercicio 3.21: *Obsérvese que hemos utilizado el par de registros D y E para albergar el multiplicando. ¿Cómo sería el programa anterior si hubiésemos utilizado el par B y C? (Una pista: sería necesario hacer una modificación al final.)*

Ejercicio 3.22: *¿Por qué hay que molestarse en poner a 0 el registro D al cargar MPD en E?*

Por último, vamos a prestar atención a un detalle que puede parecer irritante al programador no familiarizado con el Z80. Como habrá observado el lector, para cargar MPD en E a partir de la memoria es preciso cargar simultáneamente los dos registros D y E desde la dirección de memoria, porque, a menos que la dirección esté contenida en los registros H y L, no hay forma de traer un solo byte directamente y cargarlo en el

registro E; es una peculiaridad heredada del primitivo 8008, que carecía de direccionamiento directo. Con algunas mejoras, esta peculiaridad pasó al 8080, y mejoró todavía más en el Z80, en el que pueden traerse directamente 16 bits desde una dirección dada, pero no 8 bits, salvo hacia el registro A.

Ahora, una vez solucionado este problema un tanto misterioso, pasaremos a programar una multiplicación más complicada.

MULTIPLICACION DE 16 × 16

Para poner a prueba todo lo que ya hemos aprendido, vamos a multiplicar dos números de 16 bits, aunque supondremos que el resultado precisa únicamente 16 bits para que quepa en un par de registros.

Este, como en el primer ejemplo de multiplicación, pasará a los registros H y L (véase la figura 3.28). El multiplicando MPD reside en los registros D y E.

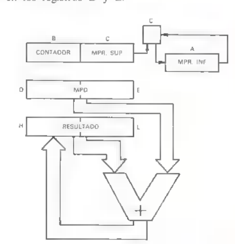


Figura 3.28
Registros de la multiplicación de 16 × 16.

Figura 3.29
Programa de multiplicaci
16 × 16.

Es tentador situar el multiplicador en los registros B y C, pero si queremos aprovechar la instrucción DJNZ, el registro B debe reservarse para el contador; en consecuencia, la mitad del multiplicador estará en el registro C, y la otra mitad, en el A (véase la figura 3.28). El programa de multiplicación aparece en la figura 3.29.

ada del primitivo 8008, que
Con algunas mejoras, esa
todavia más en el Z80, en
16 bits desde una dirección
el registro A.
problema un tanto miste-
multiplicación más compli-

que ya hemos aprendido,
16 bits, aunque supondre-
ente 16 bits para que quepa

de multiplicación, pasará a
28). El multiplicando MPD



lor en los registros B y C.
rucción DJNZ, el registro B
consecuencia, la mitad del
C, y la otra mitad, en el A
e multiplicación aparece en

```
MUL16   LD      A, (MPRAD + 1)    MPR, SUPERIOR
        LD      C, A
        LD      A, (MPRAD)        MPR, INFERIOR
        LD      B, 16D            CONTADOR
        LD      DE, (MPDAD)       MPD
        LD      HL, 0
MULT    SRL     C                 DESPLAZAMIENTO DERECHA MPR, SUPERIOR
        RRA                       ROTACION CIRCULAR DERECHA MPR, INFERIOR
        JR      NC, NOADD         VERIFICAR ACARREO
        ADD     HL, DE            SUMAR MPD AL RESULTADO
NOADD   EX      DE, HL
        ADD     HL, HL            DOBLE DESPLAZAMIENTO MPD IZQUIERDA
        EX      DE, HL
        DJNZ    MULT
        RET
```

Figura 3.29
Programa de multiplicación de 16 × 16.

El programa es análogo al que hemos desarrollado antes. Las primeras seis instrucciones (desde la etiqueta MUL16 a la MULT) inician los registros con los contenidos necesarios. El que las dos mitades de MPR deban cargarse en operaciones separadas constituye una complicación adicional. Se supone que MPRAD señala la parte inferior de MPR en la memoria; la parte superior ocupa la posición secuencial siguiente (naturalmente, puede utilizarse la convención contraria). Una vez leida la parte superior de MPR en A, debe transferirse a C:

```
LD  A,(MPRAD + 1)
LD  C,A
```

Por último, la parte inferior de MPR puede leerse directamente en el acumulador:

```
LD   A,(MPRAD)
```

El resto de los registros —B, D, E, H y L— se inician de la forma habitual:

```
LD   B,16D
LD   DE,(MPDAD)
LD   HL,0
```

Hay que realizar un desplazamiento de 16 bits sobre el multiplicador, lo que exige dos operaciones independientes de desplazamiento o de rotación circular sobre los registros C y A:

```
MULT  SLR  C
      RRA
```

Tras el desplazamiento de 16 bits, el bit de la derecha de MPR, es decir, el BmS (bit menos significativo), ocupa el bit de acarreo C, en el que puede verificarse:

```
JR   NC,NOADD
```

Como es habitual, el multiplicando no se suma al resultado si el bit de acarreo es "0", pero sí se añade si es "1".

```
ADD  HL,DE
```

A continuación hay que desplazar el multiplicando MPD una posición hacia la izquierda.

Sin embargo, el Z80 no dispone de ninguna instrucción que permita desplazar simultáneamente los contenidos de los registros D y E una posición a la izquierda, y tampoco es posible sumar a sí mismos esos contenidos, que, por tanto, deben transferirse a H y L, duplicarse y devolverse otra vez a D y E. De todo esto se encargan las tres instrucciones siguientes:

```
NOADD  EX   DE,HL
       ADD  HL,HL
       EX   DE,HL
```

Por último, se reduce el contador B y se produce un salto al principio del bucle si esa reducción no lo lleva a "0".

```
DJNZ  MULT
```

Figura 3.30
Multiplicación de 16 × 16 resultado de 32 bits.

División bina

Como siempre, pueden pensarse otras formas de organizar los registros para obtener, o no obtener, programas más cortos.

Ejercicio 3.23: *Cargar el multiplicador en los registros B y C, colocar el contador en A, escribir el programa de multiplicación correspondiente y discutir las ventajas o inconvenientes de esta organización de los registros.*

Ejercicio 3.24: *En el programa original de multiplicación de 16 bits de la figura 3.29, ¿habría alguna forma de desplazar MPD, contenido en los registros D y E, sin transferirlo a los H y L?*

Ejercicio 3.25: *Escríbase un programa de multiplicación de 16 × 16 bits que detecte resultados de más de 16 bits. Se trata de una sencilla mejora del programa básico.*

Ejercicio 3.26: *Escríbase un programa de multiplicación de 16 × 16 bits que admita resultados de 32 bits. La organización de registros sugerida aparece en la figura 3.30. Recuérdese que el resultado inicial tras la primera suma del bucle sólo necesitará 16 bits y que el multiplicador dejará un bit libre por cada iteración posterior.*

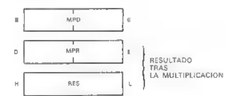


Figura 3.30
Multiplicación de 16 × 16 con resultado de 32 bits.

Pasemos ahora a la última de las operaciones aritméticas usuales: la división.

División binaria

El algoritmo de la división binaria es análogo al utilizado para la multiplicación: el divisor se resta, sucesivamente, de los bits de orden superior del dividendo; tras cada resta, se usa el resultado en lugar del dividendo inicial; simultáneamente, se incrementa cada vez en 1 el valor del cociente. A veces, el resultado de la resta es negativo, y a esa situación se le llama *sobrepasamiento;* para solucionarla, se restaura el resultado parcial, sumándole el divisor otra vez (naturalmente, hay que reducir simultáneamente en 1 el cociente). A continuación se desplazan un bit a la izquierda el cociente y el dividendo, y se repite el algoritmo. El diagrama de flujo se ilustra en la figura 3.31.

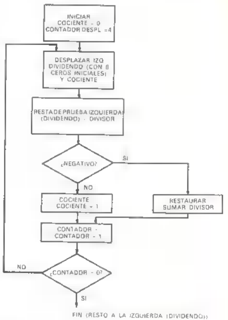


Figura 3.31
Diagrama de flujo de la división binaria de 8 bits.

Este método es el llamado de *restauración*. Hay una variante de ejecución más rápida llamada *sin restauración*.

DIVISION 16 POR 8

Examinemos, como ejemplo, una división de 16 × 8, que deja un cociente de 8 bits y un resto del mismo formato. La figura 3.32 recoge la disposición de los registros.

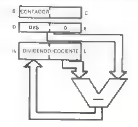


Figura 3.32
Registros de la división 16/8.

Figura 3.33
Programa de división 16

ZQUIERDA (DIVIDENDO))

staurución. Hay una variante *sin restauración*.

[illegible] división de 16 × 8, que
sto del mismo formato. La
e los registros.



El programa aparece en la figura 3.33:

```
DIV168  LD    A,(DVSAD)   CARGAR DIVI-
                          SOR
        LD    D,A         EN D
        LD    E,0
        LD    HL,(DVDAD)  CARGAR DIVI-
                          DENDO DE 16
                          BITS
        LD    B,8         INICIALIZAR
                          EL CONTADOR
DIV     XOR   A           BORRAR BIT C
        SBC   HL,DE       DIVIDEN-
                          DO - DIVISOR
        INC   HL          COCIENTE =
                          COCIENTE + 1
        JP    P,NOADD     VERIFICAR SI
                          EL RESTO ES
                          POSITIVO
        ADD   HL,DE       RESTAURAR SI
                          ES NECESARIO
        DEC   HL          COCIENTE =
                          COCIENTE - 1
NOADD   ADD   HL,HL       DESPLAZAR
                          DIVIDENDO A
                          LA IZQUIERDA
        DJNZ  DIV         BUCLE HASTA
                          QUE B = 0
        RET
```

Figura 3.33
Programa de división 16/8

Las primeras cinco instrucciones cargan el divisor y el dividendo, respectivamente, en los registros correspondientes, y, además, inicializan el contador, en el registro [illegible] al valor 8. Obsérvese que el registro B constituye el alojamiento idóneo del contador cuando se utiliza la instrucción especial del Z80, DJNZ:

```
DIV168  LD  A,(DVSAD)
        LD  D,A
        LD  E,0
        LD  HL,(DVDAD)
        LD  B,8
```

A continuación se resta el divisor del dividendo. Como hay que usar una instrucción SBC (no hay resta de 16 bits sin

acarreo), es preciso poner el acarreo a "0" antes de realizar la operación, cosa que puede hacerse de varias formas; el acarreo puede anularse con instrucciones como las siguientes:

```
XOR A
AND A
OR A
```

En este caso se ha utilizado XOR:

```
DIV  XOR  A
```

Ahora puede efectuarse la resta:

```
SBC  HL, DE
```

Se anticipa que la resta dejará un resto positivo, paso que se conoce como "resta de prueba" (véase el diagrama de flujo de la figura 3.31); por tanto, el cociente se incrementa en 1. Si la resta falla (es decir, si el resto es negativo), será preciso reducir a continuación en 1 el cociente:

```
INC  HL
```

Se verifica el resultado de la resta:

```
JP  P, NOADD
```

Si el resto es positivo o 0, es que el resultado ha sido correcto, y no es preciso almacenarlo. El programa salta a la dirección NOADD. En caso contrario, el dividendo en curso debe ponerse con su valor anterior sumándole el divisor, a la vez que se resta 1 al cociente. Las siguientes instrucciones se encargan de ejectuar estos pasos:

```
ADD  HL, DE
DEC  HL
```

Por último, se desplaza a la izquierda el dividendo resultante, como anticipación a la siguiente resta de prueba. Se reduce el contador B y se comprueba si vale "0". Mientras esto no ocurra, se ejecuta el bucle:

```
NOADD  ADD  HL, HL
       DJNZ DIV
       RET
```

Figura 3.34
Tabla para el programa de d[illegible]
sión.

o a "0" antes de realizar la
e varias formas; el acarreo
omo las siguientes:

OR:

:

n resto positivo, paso que se
se el diagrama de flujo de la
incrementa en 1. Si la resta
ivo), será preciso reducir a

esta:

; que el resultado ha sido
rlo. El programa salta a la
ario, el dividendo en curso
r sumándole el divisor, a la
s siguientes instrucciones se

uierda el dividendo resultan-
e resta de prueba. Se reduce
vale "0". Mientras esto no

Ejercicio 3.27: *Verifíquese manualmente el funcionamiento de este programa de división cumplimentando la tabla de la figura 3.34, tal como se hizo en el ejercicio 3.18 con la multiplicación. Obsérvese que no es preciso introducir en esta tabla el contenido de D, porque no se modifica nunca.*

ETIQUETA	INSTRUCCION	B	H	L

Figura 3.34
Tabla para [redacted] programa de división.

DIVISION DE 8 BITS

El programa que se propone aquí sigue un procedimiento de restauración, y deja en A un cociente complementado. Sirve para efectuar divisiones de 8 bits por [redacted] bits sin signo.

```
E ES EL DIVIDENDO
C ES EL DIVISOR
A ES EL COCIENTE
■ ES EL RESTO

DIV88    XOR   A       BORRAR EL ACU-
                       MULADOR
         LD    B,8     CONTADOR DEL
                       BUCLE
LOOP88   RL    E       PERMUTACION
                       CIRCULAR DE CY
                       EN ACC-DIVIDEN-
                       DO
         RLA           SALIDA DE CY
         SUB   C       DIVISOR DE LA
                       RESTA DE PRUE-
                       BA
```

```
JR      NC,$ + 3    RESTA CORRECT
ADD     A,C         RESTAURAR
                    ACUM, PONER C
DJNZ    LOOP88
LD      B,A         PONER EL REST
                    EN B
LD      A,E         OBTENER EL CO-
                    CIENTE
RLA                 DESPLAZAR EL
                    ULTIMO BIT DEL
                    RESULTADO
CPL                 BITS DE COM-
                    PLEMENTO
RET
```

Nota: El símbolo "$" de la sexta instrucción representa e valor del contador de programa.

DIVISION SIN RESTAURACION

El programa siguiente lleva a cabo una división de u entero de 16 bits por otro de 15 bits mediante una técnica s restauración. IX señala el dividendo e IY el divisor (no cero. (Véase la figura 3.35.)



Figura 3.35
Registros de la división sin restauración.

```
3      RESTA CORRECTA
       RESTAURAR
       ACUM, PONER CY

       PONER EL RESTO
       EN B
       OBTENER EL CO-
       CIENTE
       DESPLAZAR   EL
       ULTIMO BIT DEL
       RESULTADO
       BITS  DE  COM-
       PLEMENTO
```

ta instrucción representa el

N

cabo una división de un
its mediante una técnica sin
o e IY el divisor (no cero).

VD, INF | C

E

L

DVD

DVS

El registro ■, inicialmente de valor 16, es el contador.
A y C contienen el dividendo.
D y E contienen el divisor.
H y L contienen el resultado.

El dividendo de 16 bits se desplaza hacia la izquierda mediante las instrucciones:

```
RL    C
RLA
```

El resto se desplaza hacia la izquierda mediante la intrucción:

```
ADC   HL, HL.
```

El cociente final queda en B, C y el resto en HL. El programa continúa.

```
DIV16     LD     B,(IX + 1)
          LD     C,(IX)
          LD     D,(IY + 1)
          LD     E,(IY)
          LD     A, D
          OR     E            PARTE SUPE-
                              RIOR DEL (DI-
                              VISOR) O PAR-
                              TE  INFERIOR
                              DEL (DIVISOR)
          JR     Z, ERROR     VERIFICAR  SI
                              DIVISOR = 0
          LD     A, B         OBTIENE
                              (DVD) SUP
          LD     HL,0         BORRAR   RE-
                              SULTADO
          LD     ■, 16D       CONTADOR
TRIALSB   RL     C            ROTACION
                              CIRCULAR RE-
                              SULTADO
                              + ACC IZQ
          RLA
          ADC    HL, HL       DESPLAZAR A
                              LA  IZQUIER-
                              DA. NO PONE
                              ACARREO
          SBC    HL, DE       MENOS  DIVI-
                              SOR
```

```
NULL      CCF                   BIT DE
                                SULTADO
          JR     NC, NGV        ¿ACUMULA-
                                DOR NEGATI
                                VO?
PTV       DJNZ   TRIALSB        ¿CONTADOR
                                CERO?
          JP     DONE
RESTOR    RL     C              ROTACION
                                CIRCULAR RE
                                SULTADO
                                + ACC IZQ
          RLA
          ADC    HL, HL         COMO ARRI
                                BA
          AND    A
          ADC    HL, DE         RESTAURAR
                                SUMANDO
                                DVSR
          JR     C, PTV         RESULTADO
                                POSITIVO
          JR     Z, NULL        RESULTADO
                                CERO
NGV       DJNZ   RESTOR         ¿CONTADOR
                                CERO?
DONE      RL     C              DESPLAZAR
                                BIT DE RE
                                SULTADO
          RLA
          ADD    HL, DE         RESTO CO
                                RRECTO
          LD     B, A           COCIENTE E
                                B, C
          RET
```

Ejercicio 3.28: *Compárese el programa anterior con el siguie que utiliza una técnica de restauración:*

```
DIVIDENDO EN AC
DIVISOR EN DE
COCIENTE EN AC
RESTO EN HL

DIV16     LD     HL, 0          BORRAR ACU
                                MULADOR
```


Operaciones

```
         LD      B,16D        PONER CONTADOR
LOOP16   RL      C            ROTACION CIRCULAR ACC-RESULTADO
         RLA
         ADC     HL,HL        DESPLAZAMIENTO A LA IZQUIERDA
         SBC     HL,DE        DIVISOR RESTA DE PRUEBA
         JR      NC,$ + 3     RESTA CORRECTA
         ADD     HL,DE        RESTAURAR ACUMULADOR
         CCF                  CALCULAR BIT DEL RESULTADO
         DJNZ    LOOP16       EL CONTADOR NO ES CERO
         RL      C            DESPLAZAR EL ULTIMO BIT DEL RESULTADO
         RLA
         RET
```

Nota: El símbolo "$" de la séptima instrucción significa "posición en curso".

Operaciones lógicas

La otra clase de instrucciones que puede ejecutar la ALU son las *instrucciones lógicas:* AND, OR y OR exclusivo (XOR). También podrían incluirse aquí las operaciones de desplazamiento y rotación circular, utilizadas ya repetidamente, y la instrucción de comparación, que en el Z80 se llama CP. El uso individual de AND, OR y XOR se describirá en el capítulo 4.

Vamos ahora a desarrollar un breve programa para comprobar si una posición de memoria dada llamada LOC contiene el valor "0", el "1" o algún otro.

Utilizaremos en el programa la instrucción de comparació y realizaremos una serie de comprobaciones lógicas. Según e resultado de la comparación, se ejecutará uno u otro segment del programa.

El programa es éste:

```
               LD  A,(LOC)   LEER EL CARAC
                             TER DE LOC
               CP  00H       COMPARAR CO
                             0
               JP  Z,CERO    ¿ES A 0?
               CP  01H       COMPARAR CO
                             1
               JP  Z,UNO
NOENCONTRADO ...
CERO           ...
UNO            ...
```

La primera instrucción, "LD A,(LOC)", lee el contenido d la posición de memoria LOC y la carga en el acumulador. E contenido es el carácter que deseamos comprobar, y su valor s compara con 0 mediante la instrucción:

CP 00H

La instrucción compara el contenido del acumulador con e valor hexadecimal "00", es decir, con el binario "0000 000 Esta instrucción de comparación pone el bit Z del registro e estado al valor "1", si el resultado es afirmativo, lo que s comprueba mediante la instrucción:

JP Z,CERO

La instrucción de salto comprueba el valor del bit Z. Si resultado de la comparación ha sido positivo, Z valdrá uno y s efectuará el salto a la dirección CERO. Si el resultado es negat vo, se ejecutará la instrucción siguiente de la secuencia:

CP 01H

De la misma manera, la instrucción de salto bifurcará a l posición UNO si la comparación es positiva. Si ninguna de l comparaciones fuesen positivas, se ejecutaría la instrucción q ocupa la posición NOENCONTRADO.

```
     JP  Z,UNO
NOENCONTRADO ...
```

La finalidad de este programa es poner de relieve el valor de la instrucción de comparación seguida de salto. Esta combinación podrá utilizarse en muchos de los programas que vienen a continuación.

Ejercicio 3.29: *Búsquese en el capítulo siguiente la definición de la instrucción LD A,(LOC) y examínese su efecto sobre las banderas en caso de que hubiere alguno. ¿Es imprescindible la segunda instrucción de este programa (CP 00H)?*

Ejercicio 3.30: *Escríbase un programa que lea el contenido de la posición de memoria "24" y bifurque a la dirección llamada "ESTRELLA", si en dicha posición se encuentra el símbolo "*". La representación binaria de "*" es "00101010".*

Resumen de instrucciones

Hemos estudiado casi todas las instrucciones importantes del Z80 utilizándolas: hemos transferido valores entre la memoria y los registros; hemos realizado operaciones aritméticas y lógicas con esos valores; los hemos verificado, y, según el resultado obtenido, hemos ejecutado unas u otras porciones del programa. También hemos aprovechado las instrucciones "automáticas" especiales del Z80, como DJNZ, para acortar programas. Más adelante recurriremos a otras instrucciones automáticas, como LDDR, CPIR o INIR.

Se ha sacado el máximo partido de las características peculiares del Z80, como las instrucciones para registros de 16 bits que simplifican los programas (téngase en cuenta que tales características no existen en el 8080, del que el Z80 es una versión optimizada).

Ya hemos hablado de una estructura llamada bucle, y a continuación estudiaremos otra muy importante: la subrutina.

Subrutinas

Conceptualmente, una subrutina no es sino un bloque de instrucciones al que el programador ha adjudicado un nombre. Desde un punto de vista práctico, toda subrutina empieza con una instrucción especial, llamada *declaración de subrutina*, que la identifica como tal al ensamblador, y termina con otra,

llamada *retorno (return)*. Veremos primero el funcionamiento de una subrutina dentro de un programa, para aprender a apreciar su valor, y a continuación pasaremos a la realización práctica de la misma.

La utilización de una subrutina se muestra en la figura 3.36. El programa principal está representado a la izquierda, y la subrutina, a la derecha. Las líneas del primero se ejecutan una tras otra hasta que aparece una instrucción "CALL SUB" o *llamada a subrutina*, que transfiere el control a ésta, de manera que la primera instrucción que se ejecuta tras CALL SUB es la primera de las que componen la subrutina en cuestión, como muestra la flecha 1 de la figura.

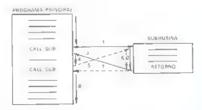


Figura 3.36
Llamadas a una subrutina.

A continuación se ejecuta el subprograma de la subrutina de la misma forma que cualquier otro programa. Supondremos en principio que la subrutina no incluye, a su vez, otras llamadas. La última instrucción de la subrutina es RETORNO, lo que provoca la devolución del control al programa principal. Tras dicha instrucción, la primera que se ejecuta es la que sigue a CALL SUB en el mencionado programa principal, hecho que muestra la flecha 3 de la figura. A continuación prosigue de la forma normal la ejecución del programa principal (flecha 4).

Dentro del programa principal surge una segunda instrucción CALL SUB, y tiene lugar una segunda transferencia, simbolizada por la flecha 5. Esto significa que la subrutina vuelve a ejecutarse, una vez más, tras la nueva llamada.

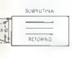
En el momento en que aparece la instrucción RET dentro de la subrutina se ejecuta la instrucción del programa principal que sigue a la CALL SUB, como muestra la flecha 7 y tras ella el resto de dicho programa (flecha 8).

El efecto de las instrucciones especiales CALL SUB y RET está, por tanto, claro. Falta por saber qué utilidad tienen las subrutinas.

Lo más valioso de una subrutina es que puede llamarse desde cualquier punto del programa principal y utilizarse cuantas veces sea necesario *sin necesidad de volver a escribirla*. De

primero el funcionamiento
programa, para aprender a
pasaremos a la realización

se muestra en la figura 3.36,
entado a la izquierda, y la
del primero se ejecutan una
nstrucción "CALL SUB" o
el control a ésta, de manera
jecuta tras CALL SUB es la
ubrutina en cuestión, como



programa de la subrutina de
programa. Supondremos en
ye, a su vez, otras llamadas.
tina es RETORNO, lo que
al programa principal. Tras
e ejecuta es la que sigue a
grama principal, hecho que
continuación prosigue de la
grama principal (flecha 4).
surge una segunda instruc-
segunda transferencia, sim-
ca que la subrutina vuelve a
eva llamada.
arece la instrucción RET
a instrucción del programa
3, como muestra la flecha 7,
ama (flecha 8).
peciales CALL SUB y RET
ber qué utilidad tienen las

ina es que puede llamarse
principal y utilizarse cuan-
d de volver a escribirla. De

esta forma se ahorra espacio de memoria y tiempo de programación, con la consiguiente simplificación del diseño de programas.

Ejercicio 3.31: *¿Cuál es el principal inconveniente de la subrutina?*

Respuesta: El inconveniente del trabajo con subrutinas se deduce fácilmente con sólo examinar el flujo de control entre ellas y el programa principal: *la velocidad general de ejecución es más baja*, porque es preciso ejecutar las instrucciones especiales CALL SUB y RETURN.

REALIZACION PRACTICA DEL MECANISMO DE LA SUBRUTINA

Vamos a ver de qué forma se tratan en el interior del microprocesador las dos instrucciones especiales CALL SUB y RET. El efecto de CALL SUB es tomar la instrucción siguiente de una nueva dirección. Como se recordará (y si no se recuerda deberá repasarse el capítulo 1), la dirección de la instrucción que debe ejecutarse a continuación de la que está en curso se encuentra en el contador del programa PC. De esto se deduce que CALL SUB modifica el contenido del PC, en el que carga la dirección de comienzo de la subrutina. Pero, *¿basta con eso?*

Para responder a esa pregunta, consideremos la segunda de las instrucciones especiales: RET. Esta instrucción determina la vuelta a la instrucción que sigue a CALL SUB, lo que sólo es posible si su dirección se ha conservado en algún sitio. Dicha dirección es el valor del contador del programa en el momento en que se llega a CALL SUB, porque el contador del programa se incrementa automáticamente cada vez que se usa (repasar el capítulo 1). Esta es precisamente la instrucción que debe conservarse para ejecutar más adelante RET.

El problema que se plantea es dónde conservar esa dirección de retorno, que debe permanecer en un sitio en el que no pueda borrarse de ninguna manera.

Antes de seguir, vamos a analizar la situación que plantea la figura 3.37: la subrutina 1 contiene una llamada a otra subrutina 2 o SUB2. El mecanismo expuesto debe funcionar también en este caso. Naturalmente, el número de llamadas internas no tiene por qué estar limitado a dos, y puede ser cualquiera N. En general, cada vez que se encuentre una nueva llamada CALL, el mecanismo de ejecución debe volver a almacenar el contador

del programa, lo que significa que hacen falta al menos 3 posiciones de memoria para este mecanismo. Además, hay qu volver, primero, desde SUB2 y, a continuación, desde SUB1. E otras palabras, lo que hace falta es una estructura capaz d conservar el orden cronológico en que se han almacenado la direcciones.

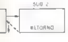
Dicha estructura tiene un nombre, y ya hemos hablado d ella: es *la pila*. La figura 3.39 recoge el contenido real de la pi durante las sucesivas llamadas a las subrutinas. Examinemo primero el programa principal. La primera llamada se encue tra en la dirección 100: CALL SUB1. Supongamos que, en e microprocesador, la llamada a la subrutina utiliza 3 bytes (RS es una excepción): por tanto, la siguiente dirección secuencia no es "101", sino "103"; la instrucción CALL utiliza las dire ciones "100", "101" y "102"; como la unidad de control del Z8 "sabe" que se trata de una instrucción de 3 bytes, el valor d contador del programa cuando la llamada haya sido decodific da en su totalidad será "103". El efecto de la llamada se cargar el valor "280" —dirección de partida de SUB1— en contador del programa.

Figura 3.37
Llamadas internas.

Figura 3.38
Llamadas a subrutinas.

Ya estamos en condiciones de estudiar el efecto de la ins trucción RET y el funcionamiento del mecanismo de l pila. La ejecución avanza dentro de SUB2 hasta que encuen tra la instrucción RET en el momento 3. El efecto de no es sino transferir la cabecera de la pila al contador d programa. En otras palabras, el contador recupera el valo que tenía antes de la entrada a la subrutina. La parte superio de la pila es, en nuestro ejemplo, "303". La figura 3.39 indic que, en el momento 3, el valor "303" ha pasado de la pila a contador del programa. Como resultado, la ejecución de l instrucción avanza a partir de la dirección "303". En el ciclo se encuentra la instrucción RET de SUB1. El valor de l cabecera de la pila es "103", que pasa al contador del progra ma. Como consecuencia, el programa se ejecuta, a partir de l posición de memoria "103", dentro del programa principal, q es precisamente el efecto deseado. La figura 3.39 demuestra q en el ciclo 4 la pila está de nuevo vacía. El mecanismo funcion

Figura 3.39
Estado de la pila a lo larg
tiempo.



Este mecanismo de llamada a subrutinas actúa hasta que la pila alcanza su dimensión máxima, y por eso los primitivos microprocesadores con pilas de 4 u 8 registros estaban limitados a 4 u 8 niveles de llamadas a subrutinas.

Obsérvese que en las figuras 3.37 y 3.38 las subrutinas se han simbolizado a la derecha del programa principal; ello obedece únicamente a razones de claridad de la representación, porque las subrutinas se escriben exactamente igual que las instrucciones normales del programa. En la hoja de papel en que aparece el listado completo del programa, las subrutinas pueden colocarse al principio del texto, en el centro del mismo o al final, y se identifican por la declaración de subrutina que las precede. Las instrucciones especiales indican al ensamblador que lo que sigue debe tratarse como una subrutina. Estas *seudoinstrucciones* del ensamblador se estudiarán en el capítulo 10.

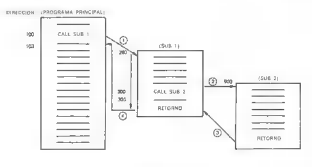


Figura 3.38
Llamadas a subrutinas.

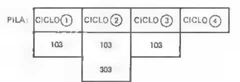


Figura 3.39
Estado de la pila a lo largo del tiempo.

SUBRUTINAS DEL Z80

Ya se han expuesto los conceptos básicos relativos a las subrutinas. Sabemos que hace falta una pila para que funcionen. El Z80 dispone de un puntero de pila de 16 bits, de manera que la pila puede residir en cualquier lugar de la memoria y albergar hasta 64K (1K = 1024), suponiendo que

estén disponibles para ese fin. En la práctica, el programador define, antes de escribir el programa, la dirección de partida de la pila y su dimensión máxima, reservándose, en consecuencia la parte necesaria de la memoria.

La instrucción de llamada a subrutinas del Z80 es CALL, y existe en dos versiones: llamada directa [redacted] incondicional —CALL DIRECCION—, que [redacted] la que ya se ha descrito, y llamada condicional, peculiar del Z80, y en virtud de la cual se llama una subrutina [redacted] se satisface determinada condición. Por ejemplo, CALL NZ, SUB1 llamará la subrutina 1 si la bandera Z es 0 en el momento de la verificación. Es una instrucción potente, porque muchas llamadas [redacted] subrutinas son condiciona-les, es decir, sólo se producen si se cumple una condición específica.

CALL CC, NN sólo s[redacted] ejecuta si es cierta la condición especificada por "CC"; CC es un conjunto de tres bits (bits 3, 4 y 5 del código de operación) capaz de especificar hasta ocho condiciones, que corresponden a cada una de las cuatro bande-ras "Z", "C", "P/V" y "S", que pueden ser cero [redacted] no cero.

Hay también dos tipos de instrucciones de vuelta: RET y RET CC.

RET es la instrucción de vuelta básica. Ocupa un byte, y hace que los dos bytes superiores de la pila vuelvan a instalarse en el contador del programa. Es incondicional.

RET CC tiene el mismo efecto, pero sólo se ejecuta [redacted] las condiciones especificadas por CC son ciertas. Los bits condicio-nales son los mismos de la instrucción CALL que acaban de describirse.

Además, hay dos tipos especializados de retorno que sirven para acabar rutinas de interrupción: RETI y RETN. Se descri-birán en el capítulo de instrucciones del Z80 y en el de interrup-ciones.

Hay, por fin, otra instrucción especializada análoga [redacted] llamada a subrutina, pero que sólo permite al programa des-viarse a una de ocho posiciones de partida localizadas en la página cero. Se trata de la instrucción RST P, una instrucción de 1 byte que almacena automáticamente el contador del pro-grama en la pila y desvía el programa a la dirección especifica-da en el campo de tres bits P; éste corresponde a los bits 3, 4 y 5 de la instrucción, multiplicados por ocho.

En otras palabras, si los bits 3, 4 y 5 son "000", el salto se producirá [redacted] la posición 00H. Si son "001", el salto será [redacted] 08H, etcétera, y así hasta 111, que provoca la bifurcación a la posición 38H. La instrucción RST es muy eficaz en términos de veloci-dad, porque tiene un solo byte, aunque a cambio de saltar únicamente [redacted] ocho posiciones en la página cero; además, estas

direcciones de la página cero están separadas nada más que por ocho bytes. Se trata de una instrucción procedente del 8080 que se usa mucho para interrupciones, como se describirá en el capítulo correspondiente. No obstante, el programador puede utilizarla para cualquier otro fin, y debe considerarse como una posible llamada a una subrutina especializada.

EJEMPLOS DE SUBRUTINAS

Casi todos los programas desarrollados hasta el momento, y la mayor parte de los que vamos a desarrollar, se escribirían normalmente como subrutinas. Así, el programa de multiplicación es normal que se use en muchos puntos de un programa general; por tanto, para clarificar y facilitar el desarrollo de programas, conviene definir una subrutina llamada, por ejemplo, MULT; al final de la misma no hay más que añadir la instrucción RET.

Ejercicio 3.32: *Si se usa MULT como subrutina, ¿"dañará" algunos de los registros o banderas internos?*

RECURRENCIA

Se llama recurrencia a la llamada a una subrutina desde ella misma. Si se ha comprendido el mecanismo de ejecución práctica de subrutinas, podrá responderse a la siguiente pregunta:

Ejercicio 3.33: *¿Es posible que una subrutina se llame a sí misma? (En otras palabras, ¿funcionará todo correctamente si una subrutina se llama a sí misma?) Si no está seguro de la respuesta, dibuje la pila y ocúpela con las direcciones sucesivas; observe a continuación los registros y la memoria (véase ejercicio 3.18) y determine si hay algún problema.*

Las interrupciones se estudiarán en el capítulo 6, dedicado a las técnicas de entrada y salida. Todos los retornos, con excepción de los que proceden de interrupciones, son instrucciones de un byte; por su parte, todas las llamadas —excepto RST— son instrucciones de tres bytes.

Ejercicio 3.34: *Consulte en el capítulo siguiente los tiempos de ejecución de las instrucciones CALL y RET. ¿Por qué el retorno de una subrutina es mucho más rápido que la llamada a la misma? (Una pista: si la respuesta no parece obvia, repásese una vez más el funcionamiento de la pila del mecanismo de subrutinas y analícense las operaciones internas que deben llevarse a cabo.)*

PARAMETROS DE SUBRUTINAS

Cuando se llama a una subrutina, normalmente se espera que actúe sobre ciertos datos. Así, en el caso de la multiplicación, hay que transmitir a la subrutina dos números para que sean sometidos a esa operación. Como ya vimos en el caso de la rutina de multiplicación, el multiplicando y el multiplicador se encuentran en posiciones de memoria dadas. He aquí, pues, un procedimiento de paso de parámetros: a través de la memoria. Pero hay, además, otras dos técnicas, lo que da lugar a tres métodos:

1. A través de los registros.
2. A través de la memoria.
3. A través de la pila.

Usar los *registros* para transferir parámetros tiene la ventaja, suponiendo que haya registros disponibles, de que no es preciso trabajar con posiciones fijas de memoria, de manera que la subrutina es independiente de la memoria. Si se utiliza una posición de memoria fija, cualquier otro usuario del programa deberá tener mucho cuidado para seguir la misma convención y asegurarse de que esa posición está realmente libre (véase el ejercicio 3.19). Por eso, en muchos casos, se reserva un bloque de posiciones de memoria para transferir parámetros entre varias subrutinas.

Usar la *memoria* tiene la ventaja de la flexibilidad (pueden usarse más datos), pero a cambio de un rendimiento inferior y de tener que ligar la subrutina a un área fija de la memoria.

La ubicación de los parámetros en la *pila* tiene la misma ventaja que el uso de los registros: la independencia de la memoria. La subrutina "sabe" que recibirá, por ejemplo, dos parámetros almacenados en la cabecera de la pila. Por supuesto, también tiene inconvenientes: la pila se satura de datos, con la consiguiente reducción del número de niveles de llamadas a subrutinas. También complica considerablemente el manejo de la pila, y puede exigir el empleo de varias de estas estructuras de datos.

La elección es responsabilidad del programador; pero, en general, se prefiere conservar la mayor independencia posible con respecto a las posiciones reales de memoria.

Si no hay registros disponibles, la pila es una alternativa a considerar. Sin embargo, cuando es necesario pasar a la subrutina mucha información, ésta puede residir directamente en memoria. Una forma elegante de resolver el problema de transmitir un bloque de datos consiste simplemente en transmitir un puntero dirigido a la información (un puntero es la direc-

ción del principio del bloque). El puntero puede pasarse por medio de un registro, o de la pila (hacen falta dos posiciones de pila para almacenar una dirección de 16 bits), o bien por medio de una o varias posiciones fijas de memoria.

Por último, si ninguna de las dos soluciones es aplicable, habrá que acordar con la subrutina una posición fija en memoria (el "buzón de correos").

Ejercicio 3.35: *¿Cuál de los tres métodos mencionados será mejor para las recurrencias?*

BIBLIOTECA DE SUBRUTINAS

La organización de las diversas porciones de un programa en forma de subrutinas identificables tiene la ventaja de que pueden ponerse a punto independientemente y de que se les puede asignar un nombre mnemotécnico. Si pueden utilizarse en segmentos diferentes del programa serán intercambiables y, por tanto, podrán formar parte de una biblioteca de subrutinas. Sin embargo, en programación no existe ninguna panacea, y acostumbrarse a convertir cualquier grupo de instrucciones que se repita por su función en una subrutina dará lugar a un rendimiento escaso. El programador deberá aprender a equilibrar las ventajas con los inconvenientes.

Resumen

Hemos visto en este capítulo cómo manipulan internamente la información las instrucciones del Z80. Los algoritmos traducidos a programas se han hecho cada vez más complicados y, además, han servido para utilizar y explicar los tipos de instrucciones más importantes.

También hemos definido varias estructuras de uso continuo, como bucles, pilas y subrutinas.

El lector deberá tener ya una idea básica de lo que es programar y de las principales técnicas puestas en juego en las aplicaciones normales. Pasemos, pues, a estudiar en detalle cada una de las instrucciones.

```
       A=00 BC=0000 DE=0000 HL=0000 S=0300 P=0100 0100' LD   BC,(03
      A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00        (0300)
       A=00 BC=0003 DE=0000 HL=0000 S=0300 P=0104 0104' LD   B,08
      A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
       A=00 BC=0803 DE=0000 HL=0000 S=0300 P=0106 0106' LD   DE,(03
      A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00        (0302)
       A=00 BC=0803 DE=0005 HL=0000 S=0300 P=010A 010A' LD   D,00
      A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
       A=00 BC=0803 DE=0005 HL=0000 S=0300 P=010C 010C' LD   HL,000
      A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00        (0000)
       A=00 BC=0803 DE=0005 HL=0000 S=0300 P=010F 010F' SRL  C
      A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
     C A=00 BC=0801 DE=0005 HL=0000 S=0300 P=0111 0111' JR   NC,011
      A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00        (0114)
     C A=00 BC=0801 DE=0005 HL=0000 S=0300 P=0113 0113' ADD  HL,DE
      A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
       A=00 BC=0801 DE=0005 HL=0005 S=0300 P=0114 0114' SLA  E
      A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
   V   A=00 BC=0801 DE=000A HL=0005 S=0300 P=0116 0116' RL   D
      A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
 Z V   A=00 BC=0801 DE=000A HL=0005 S=0300 P=0118 0118' DEC  B
      A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
     N A=00 BC=0701 DE=000A HL=0005 S=0300 P=0119 0119' JP   NZ,01
      A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00        (010F)
     N A=00 BC=0701 DE=000A HL=0005 S=0300 P=010F 010F' SRL  C
      A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
 Z V C A=00 BC=0700 DE=000A HL=0005 S=0300 P=0111 0111' JR   NC,011
      A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00        (0116)
 Z V C A=00 BC=0700 DE=000A HL=0005 S=0300 P=0113 0113' ADD  HL,DE
      A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
 Z V   A=00 BC=0700 DE=000A HL=000F S=0300 P=0114 0114' SLA  E
      A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
   V   A=00 BC=0700 DE=0014 HL=000F S=0300 P=0116 0116' RL   D
      A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
 Z V   A=00 BC=0700 DE=0014 HL=000F S=0300 P=0118 0118' DEC  B
      A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
     N A=00 BC=0600 DE=0014 HL=000F S=0300 P=0119 0119' JP   NZ,01
      A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00        (010F)
     N A=00 BC=0600 DE=0014 HL=000F S=0300 P=010F 010F' SRL  C
      A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
 Z V   A=00 BC=0600 DE=0014 HL=000F S=0300 P=0111 0111' JR   NC,011
      A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00        (0116)
 Z V   A=00 BC=0600 DE=0014 HL=000F S=0300 P=0114 0114' SLA  E
      A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
   V   A=00 BC=0600 DE=0028 HL=000F S=0300 P=0116 0116' RL   D
      A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
 Z V   A=00 BC=0600 DE=0028 HL=000F S=0300 P=0118 0118' DEC  B
      A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
     N A=00 BC=0500 DE=0028 HL=000F S=0300 P=0119 0119' JP   NZ,01
      A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00        (010F)
     N A=00 BC=0500 DE=0028 HL=000F S=0300 P=010F 010F' SRL  C
      A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
 Z V   A=00 BC=0500 DE=0028 HL=000F S=0300 P=0111 0111' JR   NC,011
      A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00        (0116)
 Z V   A=00 BC=0500 DE=0028 HL=000F S=0300 P=0114 0114' SLA  E
      A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
   V   A=00 BC=0500 DE=0050 HL=000F S=0300 P=0116 0116' RL   D
      A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
 Z V   A=00 BC=0500 DE=0050 HL=000F S=0300 P=0118 0118' DEC  B
      A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
     N A=00 BC=0400 DE=0050 HL=000F S=0300 P=0119 0119' JP   NZ,01
      A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00        (010F)
     N A=00 BC=0400 DE=0050 HL=000F S=0300 P=010F 010F' SRL  C
      A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
 Z V   A=00 BC=0400 DE=0050 HL=000F S=0300 P=0111 0111' JR   NC,011
      A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00        (0116)
 Z V   A=00 BC=0400 DE=0050 HL=000F S=0300 P=0114 0114' SLA  E
      A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
S  V   A=00 BC=0400 DE=00A0 HL=000F S=0300 P=0116 0116' RL   D
      A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
 Z V   A=00 BC=0400 DE=00A0 HL=000F S=0300 P=0118 0118' DEC  B
      A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
     N A=00 BC=0300 DE=00A0 HL=000F S=0300 P=0119 0119' JP   NZ,01
      A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00        (010F)
```

Figura 3.40
Listado completo de la multiplicación.

Figura 3.40
Listado completo de la m... cación (continuación).

Figura 3.41
Programa de multiplic... (Hex).

```
300 P=0100 0100' LD   BC,(0200)
000 Y=0000 I=00       (0200')
300 P=0104 0104' LD   B,08
000 Y=0000 I=00
300 P=0106 0106' LD   DE,(0202)
000 Y=0000 I=00       (0202')
300 P=010A 010A' LD   D,00
000 Y=0000 I=00
300 P=010C 010C' LD   HL,0000
000 Y=0000 I=00       (0000')
300 P=010F 010F' SRL  C
000 Y=0000 I=00
300 P=0111 0111' JR   NC,0114
000 Y=0000 I=00       (0114')
300 P=0113 0113' ADD  HL,DE
000 Y=0000 I=00
300 P=0114 0114' SLA  E
000 Y=0000 I=00
300 P=0116 0116' RL   D
000 Y=0000 I=00
300 P=0118 0118' DEC  B
000 Y=0000 I=00
300 P=0119 0119' JP   NZ,010F
000 Y=0000 I=00       (010F')
300 P=010F 010F' SRL  C
000 Y=0000 I=00
300 P=0111 0111' JR   NC,0114
000 Y=0000 I=00       (0114')
300 P=0113 0113' ADD  HL,DE
000 Y=0000 I=00
300 P=0114 0114' SLA  E
000 Y=0000 I=00
300 P=0116 0116' RL   D
000 Y=0000 I=00
300 P=0118 0118' DEC  B
000 Y=0000 I=00
300 P=0119 0119' JP   NZ,010F
000 Y=0000 I=00       (010F')
300 P=010F 010F' SRL  C
000 Y=0000 I=00
300 P=0111 0111' JR   NC,0114
000 Y=0000 I=00       (0114')
300 P=0114 0114' SLA  E
000 Y=0000 I=00
300 P=0116 0116' RL   D
000 Y=0000 I=00
300 P=0118 0118' DEC  B
000 Y=0000 I=00
300 P=0119 0119' JP   NZ,010F
000 Y=0000 I=00       (010F')
300 P=010F 010F' SRL  C
000 Y=0000 I=00
300 P=0111 0111' JR   NC,0114
000 Y=0000 I=00       (0114')
300 P=0114 0114' SLA  E
000 Y=0000 I=00
300 P=0116 0116' RL   D
000 Y=0000 I=00
300 P=0118 0118' DEC  B
000 Y=0000 I=00
300 P=0119 0119' JP   NZ,010F
000 Y=0000 I=00       (010F')
300 P=010F 010F' SRL  C
000 Y=0000 I=00
0300 P=0111 0111' JR  NC,0114
0000 Y=0000 I=00      (0114')
0300 P=0114 0114' SLA E
0000 Y=0000 I=00
0300 P=0116 0116' RL  D
0000 Y=0000 I=00
0300 P=0118 0118' DEC B
0000 Y=0000 I=00
0300 P=0119 0119' JP  NZ,010F
0000 Y=0000 I=00      (010F')
```

Figura 3.40
Listado completo de la multiplicación (continuación).

```
  N     A=00  BC=0300 DE=00A0 HL=000F S=0300 P=010F 010F' SRL  C
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V     A=00  BC=0300 DE=00A0 HL=000F S=0300 P=0111 0111' JR   NC,0114
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00       (0114')
Z V     A=00  BC=0300 DE=00A0 HL=000F S=0300 P=0114 0114' SLA  E
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
    C   A=00  BC=0300 DE=0040 HL=000F S=0300 P=0116 0116' RL   D
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
        A=00  BC=0300 DE=0140 HL=000F S=0300 P=0118 0118' DEC  B
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
  N     A=00  BC=0200 DE=0140 HL=000F S=0300 P=0119 0119' JP   NZ,010F
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00       (010F')
  N     A=00  BC=0200 DE=0140 HL=000F S=0300 P=010F 010F' SRL  C
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V     A=00  BC=0200 DE=0140 HL=000F S=0300 P=0111 0111' JR   NC,0114
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00       (0114')
Z V     A=00  BC=0200 DE=0140 HL=000F S=0300 P=0114 0114' SLA  E
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
S       A=00  BC=0200 DE=0180 HL=000F S=0300 P=0116 0116' RL   D
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
        A=00  BC=0200 DE=0280 HL=000F S=0300 P=0118 0118' DEC  B
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
  N     A=00  BC=0100 DE=0280 HL=000F S=0300 P=0119 0119' JP   NZ,010F
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00       (010F')
  N     A=00  BC=0100 DE=0280 HL=000F S=0300 P=010F 010F' SRL  C
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V     A=00  BC=0100 DE=0280 HL=000F S=0300 P=0111 0111' JR   NC,0114
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00       (0114')
Z V     A=00  BC=0100 DE=0280 HL=000F S=0300 P=0114 0114' SLA  E
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V C   A=00  BC=0100 DE=0200 HL=000F S=0300 P=0116 0116' RL   D
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
  V     A=00  BC=0100 DE=0500 HL=000F S=0300 P=0118 0118' DEC  B
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z N     A=00  BC=0000 DE=0500 HL=000F S=0300 P=0119 0119' JP   NZ,010F
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00       (010F')
Z N     A=00  BC=0000 DE=0500 HL=000F S=0300 P=011C 011C' LD   (0204),HL
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00       (0204')
Z N     A=00  BC=0000 DE=0500 HL=000F S=0300 P=011F 011F' NOP
        A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
```

RESPUESTAS AL EJERCICIO 3.18 (MULTIPLICACION):

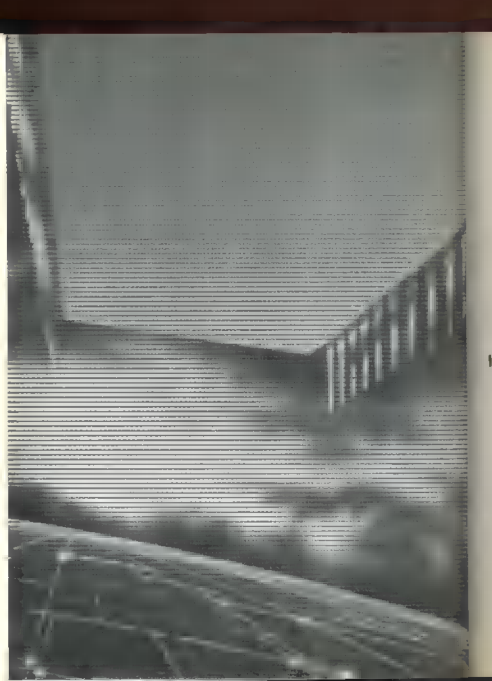
```
0100            10        ORG   #0100
0100 0002       20 MPRAD  DEFW  #0200
0102 0202       30 MPDAD  DEFW  #0202
0104 0402       40 RESAD  DEFW  #0204
                50 ;
0106 ED4B0001   60 MP488  LD    BC,(MPRAD)    ;CARGA MULTIPLICADOR EN C
010A 0608       70        LD    B,8           ;B ES CONTADOR DE BIT
010C ED5B0201   80        LD    DE,(MPDAD)    ;CARGA MULTIPLICANDO EN E
0110 1600       90        LD    D,0           ;INICIALIZA D
0112 210000    100        LD    HL,0          ;PONE A 0 EL RESULTADO
0115 CB39      110 MULT   SRL   C             ;SHIFT AL ACARREO DEL BIT
               115 ;                          ;MULTIPLICADOR
0117 3001      120        JR    NC,NOADD      ;COMPRUEBA EL ACAREO
0119 19        130        ADD   HL,DE         ;SUMA AL RESULTADO MPD
011A CB23      140 NOADD  SLA   E             ;SHIFT IZQUIERDA DE MPD
011C CB12      150        RL    D             ;GUARDA EL BIT EN D
011E 05        160        DEC   B             ;DECREMENTA EL CONTADOR DE SHIFT
011F C21501    170        JP    NZ,MULT       ;REPETIR SI CONTADOR<>0
0122 220401    180        LD    (RESAD),HL    ;ALMACENA EL RESULTADO
```

Figura 3.41
Programa de multiplicación (Hex).

ETIQUETA	INSTRUCCION	B	C	C (ACARREO)	[illegible]	E	H	L
		00	00	[illegible]	00	00	00	00
MP488	LD BC,(0200)	00	03	0	00	00	00	00
	LD B, 08	08	03	0	00	00	00	[illegible]
	LD DE,(0202)	08	03	0	00	05	00	00
	LD D, 00	08	03	0	00	05	00	00
	LD HL,0000	08	03	0	00	05	00	00
MULT	SRL C	08	01	1	00	05	00	00
	JR NC,0114	08	01	1	00	05	00	00
	ADD HL,DE	08	01	[illegible]	00	05	00	05
NOADD	SLA E	08	01	0	00	0A	00	05
	RL D	08	01	0	00	0A	00	05
	DEC B	07	01	0	00	0A	00	05
	JP NZ,010F	07	01	[illegible]	00	0A	00	[illegible]
MULT	SRL C	07	00	1	00	0A	00	05
	JR NC,0114	07	00	1	00	0A	00	05
	ADD HL,DE	07	00	0	00	0A	00	0F
NOADD	SLA E	07	00	0	00	14	00	0F
	RL D	07	00	0	00	14	00	0F
	DEC B	06	00	0	00	14	00	0F
	JP NZ,010F	06	00	0	00	14	00	0F

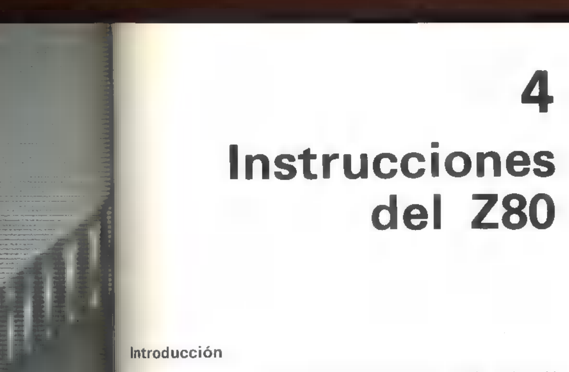
Figura 3.42
Dos repeticiones del bucle.

C (ACARREO)	D	E	H	L
0	00	00	00	00
0	00	00	00	00
0	00	00	00	00
0	00	05	00	00
0	00	05	00	00
0	00	05	00	00
1	00	05	00	00
1	00	05	00	00
0	00	05	00	05
0	00	0A	00	05
0	00	0A	00	05
0	00	0A	00	05
0	00	0A	00	05
1	00	0A	00	05
1	00	0A	00	05
0	00	0A	00	0F
0	00	14	00	0F
0	00	14	00	0F
0	00	14	00	0F
0	00	14	00	0F



Introducció

Clases de



4

Instrucciones del Z80

Introducción

Antes de analizar una por una todas las instrucciones del Z80 y de explicar en detalle su finalidad, el efecto que provocan en las banderas *(flags)* y cómo pueden combinarse con los diversos modos de direccionamiento, estudiaremos los tipos de aquellas que deben existir en cualquier ordenador de tipo general. El capitulo 5 está íntegramente dedicado a la discusión en profundidad de las técnicas de direccionamiento.

Clases de instrucciones

Las instrucciones pueden clasificarse con arreglo a muchos criterios. En este libro reconoceremos las siguientes cinco categorias:

1. Transferencia de datos.
2. Tratamiento de datos.
3. Verificación y bifurcación.
4. Entrada y salida.
5. Control.

Vamos ahora a describirlas una por una.

TRANSFERENCIA DE DATOS

Estas instrucciones llevan datos de unos registros a otros entre los registros y la memoria o entre los registros y algún dispositivo de entrada/salida. Puede haber instrucciones de transferencia específicas para registros especializados; así, hay operaciones de inserción (*push*) y extracción (*pop*), para manejar la pila con más eficacia; cargan una palabra de datos de la posición superior de la pila en el acumulador, y actualizan automáticamente el registro del apuntador de la misma, todo ello en una sola instrucción.

TRATAMIENTO DE DATOS

Las instrucciones de tratamiento de datos pueden, a su vez clasificarse en cinco categorías generales:

1. Operaciones aritméticas (como adición o sustracción).
2. Manipulación de bits (SET y RESET).
3. Incremento y decremento.
4. Operaciones lógicas (AND, OR, OR exclusivo).
5. Operaciones de desplazamiento y rotación (SHIFT, ROTATE).

Para manipular los datos con eficacia es muy deseable disponer de instrucciones aritméticas potentes, como la multiplicación y división, aunque, por desgracia, están ausentes de la mayor parte de los microprocesadores. Son también muy interesantes instrucciones potentes de desplazamiento y rotación como desplazar n bits o intercambiar *nibbles* (es decir, cambiar de posición las mitades izquierda y derecha de un byte), aunque también éstas faltan de casi todos los microprocesadores.

Antes de examinar las instrucciones del Z80, recordemos la diferencia entre *desplazamiento* y *rotación*. El desplazamiento es la traslación del contenido de un registro o de una posición de memoria a la izquierda o a la derecha en un bit; como resultado sale del registro un bit, que pasa al bit de acarreo. El bit que entra por el lado opuesto será "0", salvo que se haya efectuado un "desplazamiento aritmético a la derecha", que tiene como resultado la duplicación del bit más significativo.

En la rotación, el bit que sale también pasa al acarreo; pero el que entra es el que había en dicho acarreo antes del inicio de la operación; por tanto, equivale a una permutación circular de 9 bits. Con frecuencia conviene disponer de una instrucción de permutación de 8 bits que traslade el bit de un extremo al contrario; no la tiene casi ningún microprocesador, aunque sí el Z80 (véase figura 4.1).

Figura 4.1
Desplazamiento y rotació

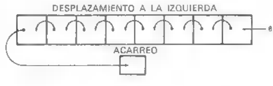




Figura 4.1
Desplazamiento y rotación.

Por último, al desplazar una palabra a la derecha es conveniente disponer de otro tipo de desplazamiento, llamado extensión de signo o "desplazamiento aritmético a la derecha". Al operar en complemento a dos, sobre todo al ejecutar rutinas de punto flotante, suele ser necesario desplazar un número negativo a la derecha; en este caso, el bit que entra por la izquierda ha de ser un "1", porque el signo debe conservarse tantas veces cuantas se efectúe el desplazamiento: esto es lo que se llama desplazamiento aritmético a la derecha.

VERIFICACION Y SALTO

Las instrucciones de verificación comprueban si los bits contenidos en el registro que se especifica son "0", "1" o alguna combinación determinada de esos valores. Como mínimo, es necesario que pueda verificarse el registro de estado, lo que significa que conviene disponer en ese registro del mayor número posible de banderas. Es aconsejable que la comprobación de tales bits pueda hacerse con una sola instrucción. Lo mejor, sin duda, sería poder verificar *cualquier bit de cualquier registro* y comparar el valor de cualesquiera otros registros (mayor que, menor que, igual a). Las instrucciones de la mayor parte de los microprocesadores sólo sirven para verificar bits aislados del registro de estado, pero el Z80 está mejor equipado.

Las instrucciones de salto de las que se suele disponer se organizan en tres categorías:

1. Salto, que especifica una dirección completa de 16 bits.
2. Salto relativo, normalmente limitado a un campo de desplazamiento de 8 bits.
3. Llamada, que se emplea en combinación con subrutinas.

Conviene disponer de saltos de dos o incluso tres salidas (mayor que, menor que o igual a, por ejemplo). Es cómodo el salto relativo, que es el salto hacia adelante o hacia atrás de unas pocas instrucciones, aunque, en realidad, equivale al salto normal. Al término de la mayor parte de los bucles hay una operación de incremento o decremento seguida por otra de verificación y bifurcación; por tanto, la disponibilidad de todas estas operaciones en una sola instrucción influye decisivamente en la ejecución eficaz de bucles. Lamentablemente, la mayor parte de los microprocesadores sólo disponen de bifurcaciones sencillas combinadas con verificaciones igualmente sencillas, lo que, naturalmente, complica la programación y reduce la eficacia. El Z80 sí dispone de una instrucción de "decremento y salto", aunque sólo comprueba si un registro determinado (B) vale 0.

ENTRADA/SALIDA

Las instrucciones de entrada/salida sirven para manipular los dispositivos de entrada/salida. La mayoría de los microprocesadores de 8 bits trabajan con *E/S direccionada en memoria*; esto implica que los dispositivos de entrada y salida están conectados al *bus* de direcciones igual que las pastillas de memoria y se direccionan de la misma forma. A efectos de programación se tratan como posiciones de memoria que, por lo general, necesitan 3 bytes y, en consecuencia, son lentas. Lo mejor para aumentar la eficacia es contar con un mecanismo de direccionamiento breve para que los dispositivos de E/S, en los que la velocidad es crucial, puedan residir en la página 0. Sin embargo, si la página 0 tiene direccionamiento, suele ser, por lo general, para la memoria RAM, lo que impide su utilización eficaz para los dispositivos de E/S. El Z80, como el 8080, dispone de instrucciones especiales de entrada y salida, de manera que el programador puede direccionar los dispositivos de E/S como posiciones de memoria o como tales, utilizando para ello instrucciones E/S, que describiremos más adelante en este mismo capítulo.

INSTRUCCIONES DE CONTROL

Las instrucciones de control proporcionan las señales de sincronización y pueden suspender o interrumpir un programa. También funcionan como paradas o interrupciones simuladas (las interrupciones se tratarán en el capítulo 6, dedicado a las técnicas de entrada y salida).

e dos o incluso tres salidas
por ejemplo). Es cómodo el
a adelante o hacia atrás de
en realidad, equivale al salto
parte de los bucles hay una
mento seguida por otra de
o, la disponibilidad de todas
ucción influye decisivamente
Lamentablemente, la mayor
o disponen de bifurcaciones
ones igualmente sencillas, lo
gramación y reduce la efica-
strucción de "decremento y
un registro determinado (B)

alida sirven para manipular
La mayoría de los micropro-
/S direccionada en memoria;
de entrada y salida están
igual que las pastillas de
misma forma. A efectos de
ciones de memoria que, por
consecuencia, son lentas. Lo
contar con un mecanismo de
s dispositivos de E/S, en los
residir en la página 0. Sin
ionamiento, suele ser, por lo
o que impide su utilización
S. El Z80, como el 8080,
s de entrada y salida, de
direccionar los dispositivos
ia o como tales, utilizando
cribiremos más adelante en

L

roporcionan las señales de
o interrumpir un programa,
o interrupciones simuladas
capítulo 6, dedicado a las

Las instrucciones del Z80

INTRODUCCION

El microprocesador Z80 se ha construido como sustituto del 8080, al que supera en algunos aspectos; contiene todas las instrucciones de éste y algunas nuevas. Como el código de operación de 8 bits ofrece poco espacio, asombra que quienes proyectaron el Z80 hayan conseguido añadirle más instrucciones; para ello aprovecharon unos códigos de operación sin usar que había en el 8080 y añadieron un byte nuevo al código de las operaciones indexadas. Por eso algunas instrucciones del Z80 ocupan hasta 5 bytes de memoria.

Es importante recordar que un mismo programa puede escribirse de muchas formas diferentes. Para programar con eficacia es imprescindible conocer a fondo y entender todas las instrucciones, aunque cuando se está aprendiendo no hay necesidad de escribir programas optimizados. Así es que, al acercarse por vez primera a este capítulo, lo que importa no es recordar las instrucciones en su totalidad, sino estudiar las categorías y los ejemplos típicos. Más adelante, en el momento de escribir programas, el lector podrá consultar la descripción de las instrucciones y escoger las más adecuadas a sus necesidades. En esta sección trataremos de simplificar las instrucciones del Z80 y de agruparlas en categorías lógicas. El lector interesado en explorar las posibilidades de cada una de ellas no tiene más que consultar las descripciones individuales.

Analizaremos las posibilidades del Z80 en términos de las cinco categorías de instrucciones definidas al principio de este capítulo.

INSTRUCCIONES DE TRANSFERENCIA DE DATOS

El Z80 dispone de cuatro clases de instrucciones dentro de esta categoría: para hacer transferencias de 8 bits, para hacer transferencias de 16 bits, para realizar operaciones en la pila y para transferir bloques. Examinémoslas más despacio.

TRANSFERENCIAS DE DATOS DE 8 BITS

Se hacen todas con instrucciones de carga, que tienen el formato:

LD destino, origen

Así, para cargar el acumulador A a partir del registro B, se utilizaría la instrucción

LD A, B

Pueden hacerse transferencias directas entre dos registros de trabajo cualesquiera (A, B, C, D, E, H, L).

Para cargar un registro de trabajo a partir de una posición de memoria, excepción hecha del acumulador, hay que cargar la dirección de dicha posición en el par de registros HL.

Por ejemplo, para cargar el registro C con la posición de memoria 1234, primero se cargan los registros H y L con el valor "1234", utilizando para ello una instrucción de carga de 16 bits, que describiremos en la próxima sección. A continuación se utiliza la instrucción LD C,(HL), que dará lugar al resultado que se busca.

El acumulador constituye una excepción, porque puede cargarse directamente a partir de cualquier posición especificada de la memoria. Esto se llama modo de direccionamiento extendido. Por ejemplo, para cargar el acumulador con el contenido de la posición de memoria 1234, se emplea la instrucción siguiente:

LD A,(1234H) (Obsérvese el uso de "( )" para denotar "el contenido de")

La instrucción se almacenará en la memoria como sigue:

dirección	PC	: 3A	(código de operación)
	PC + 1	: 34	(byte inferior de la dirección)
	PC + 2	: 12	(byte superior de la dirección)

Obsérvese que en la instrucción propiamente dicha la dirección se almacena en orden inverso:

3A	mitad inferior	mitad superior

Los registros de trabajo pueden también cargarse con cualquier valor especificado de 8 bits o "literal" contenido en el segundo byte de la instrucción (esto se llama *direccionamiento inmediato*). Ejemplo:

LD E, 12H

que carga en el registro E el valor hexadecimal 12.

Figura 4.2 Instrucciones de carga ... bits ("LD").

En la memoria, la instrucción aparece así:

PC : 1E (código de operación)
PC + 1 : 12 (operando literal)

Como resultado de esta instrucción aparecerá en el registro E el operando inmediato o valor literal.

También pueden cargarse registros en modo de *direccionamiento indexado*, como veremos detalladamente en el próximo capítulo. Para cargar registros específicos hay otras posibilidades, que recoge la tabla de la figura 4.2; las zonas sombreadas muestran las instrucciones comunes al 8080A.

		FUENTE															
		IMPLÍCITO		REGISTRO							REG. INDIRECTO			INDEXADO		DIR. EXT.	INMEDIATO
		I	R	A	B	C	D	E	H	L	(HL)	(BC)	(DE)	(IX+d)	(IY+d)	(nn)	n
DESTINO REGISTRO	A	ED 57	ED 5F	7F	78	79	7A	7B	7C	7D	7E	0A	1A	DD 7E d	FD 7E d	3A n n	3E n
	B			47	40	41	42	43	44	45	46			DD 46 d	FD 46 d		06 n
	C			4F	48	49	4A	4B	4C	4D	4E			DD 4E d	FD 4E d		0E n
	D			57	50	51	52	53	54	55	56			DD 56 d	FD 56 d		16 n
	E			5F	58	59	5A	5B	5C	5D	5E			DD 5E d	FD 5E d		1E n
	H			67	60	61	62	63	64	65	66			DD 66 d	FD 66 d		26 n
	L			6F	68	69	6A	6B	6C	6D	6E			DD 6E d	FD 6E d		2E n
REG. INDIRECTO	(HL)			77	70	71	72	73	74	75							36 n
	(BC)			02													
	(DE)			12													
INDEXADO	(IX+d)			DD 77 d	DD 70 d	DD 71 d	DD 72 d	DD 73 d	DD 74 d	DD 75 d							DD 36 d n
	(IY+d)			FD 77 d	FD 70 d	FD 71 d	FD 72 d	FD 73 d	FD 74 d	FD 75 d							FD 36 d n
DIR. EXT.	(nn)			32 n n													
IMPLÍCITO	I			ED 47													
	R			ED 4F													

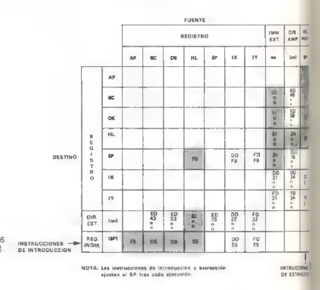
Figura 4.2
Instrucciones de carga de 8 bits ("LD").

TRANSFERENCIAS DE DATOS DE 16 BITS

En términos generales, cualquier par de registros de 16 bits, BC, DE, HL, SP, IX, IY, puede cargarse con un operando literal de 16 bits, a partir de una dirección de memoria especificada *(direccionamiento ampliado)* o a partir de la parte superior de la pila (es decir, con la dirección contenida en SP). A su vez, los contenidos de esos pares de registros pueden almacenarse de

la misma forma en cualquier posición especificada de la memoria o en la parte superior de la pila. Además, el registro SP puede cargarse a partir de HL, IX e IY, lo que facilita la creación de varias pilas. También el par de registros AF puede introducirse en la parte superior de la pila.

La tabla de la figura 4.3 recoge todas las posibilidades. Las operaciones de extracción e introducción en la pila se consideran transferencias de datos de 16 bits; dichas operaciones transfieren los contenidos de un par de registros a la pila y viceversa. Obsérvese que no hay instrucciones únicas de introducción y extracción de la pila para guardar registros individuales de 8 bits.



		FUENTE									
		REGISTRO							INM. EXT.	DIR. AMP.	[illegible]
		AF	BC	DE	HL	SP	IX	IY	nn	(nn)	[illegible]
DESTINO REGISTRO	AF										[illegible]
	BC								01 n n	ED 4B n n	[illegible]
	DE								11 n n	ED 5B n n	[illegible]
	HL								21 n n	2A n n	[illegible]
	SP				F9		DD F9	FD F9	31 n n	ED 7B n n	
	IX								DD 21 n n	DD 2A n n	[illegible]
	IY								FD 21 n n	FD 2A n n	[illegible]
DIR. EXT.	(nn)		ED 43 n n	ED 53 n n	22 n n	ED 73 n n	DD 22 n n	FD 22 n n			
REG. INDIR.	(SP)	F5	C5	D5	E5		DD E5	FD E5			

INSTRUCCIONES DE INTRODUCCION → (fila REG. INDIR.)

INTRUCCIONES DE EXTRACCIÓN

NOTA: Las instrucciones de introducción y extracción ajustan el SP tras cada ejecución.

Figura 4.3
Instrucciones de carga de 16 bits ("LD", "PUSH" y "POP").

La introducción o extracción de un doble byte se hace siempre sobre un par de registros: AF, BC, DE, HL, IX, IY (véanse la fila inferior y la columna derecha de la figura 4.3).

Para operar con AF, BC, DE y HL basta una instrucción de un byte, lo que da lugar a una eficacia considerable. Supongamos, por ejemplo, que el apuntador de la pila SP contiene el valor "0100". Se ejecuta la instrucción

PUSH AF

Figura 4.4
Intercambios "EX" y "E

Al introducir en la pila el contenido del par de registros, primero se decrementa el apuntador SP, y a continuación se deposita en la parte superior de la pila el contenido del registro A; acto seguido vuelve ■ decrementarse SP, y pasa a la pila el contenido de F. Al término de la transferencia a la pila, SP señala el elemento superior de la misma, que en nuestro ejemplo es el valor de F.

Es importante recordar que en el Z80 SP señala la parte *superior* de la pila y se *decrementa* cada vez que se carga un par de registros. Hay procesadores en los que no rige la misma convención, lo que puede causar confusiones.

INSTRUCCIONES DE INTERCAMBIO

El código mnemónico EX controla las operaciones de intercambio que, en realidad, son transferencias dobles de datos. La instrucción EX modifica los contenidos de *dos* posiciones especificadas. Así, EX puede usarse para intercambiar la parte superior de la pila con HL, IX e IY y también para intercambiar los contenidos de DE y HL y de AF y AF' (recuerde que AF' corresponde al otro par de registros AF con que cuenta el Z80).

Por último, hay una instrucción especial EXX para intercambiar los contenidos de BC, DE y HL con los de los registros correspondientes del segundo banco del Z80.

La figura 4.4 resume todos los intercambios posibles.

		DIRECCIONAMIENTO IMPLICITO				
		AF'	BC', DE' & HL'	HL	■	IY
IMPLICITO	AF	■				
	BC, DE & HL		D9			
	DE			EB		
REG. INDIR	(SP)			E3	DD E3	FD E3

Figura 4.4
Intercambios "■" y "EXX".

INSTRUCCIONES DE TRANSFERENCIA DE BLOQUES

Mediante estas instrucciones se transfiere no un byte sencillo o doble, sino un bloque completo de datos. Para el fabricante son más difíciles de realizar que casi todas las demás instrucciones, y por eso faltan en la mayor parte de los microprocesadores; pero resultan muy cómodas para el programador y aumentan la eficacia de los programas, sobre todo de las operaciones de entrada/salida. Sus aplicaciones y sus ventajas quedarán claras a lo largo del resto del libro. El Z80 dispone de algunas instrucciones automáticas de transferencia de datos que emplean convenciones específicas.

Todas esas instrucciones exigen el empleo de tres pares de registros: BC, DE, HL.

BC se utiliza como contador de 16 bits, lo que significa que pueden transferirse automáticamente hasta 2^{16} = 64K. HL actúa como apuntador fuente capaz de señalar cualquier posición de la memoria. DE es el apuntador destino y también puede señalar cualquier punto de la memoria.

Hay cuatro instrucciones de transferencia de bloques:

LDD, LDDR, LDI, LDIR

Todas ellas decrementan el registro contador BC tras cada transferencia. Dos de ellas —LDD y LDDR— decrementan los registros apuntadores DE y HL, y las otras dos —LDI y LDIR— los incrementan. En los dos grupos de instrucciones, la letra R del final del código mnemotécnico significa repetición automática. Analicemos estas instrucciones con un poco más de detalle.

LDI significa "carga e incremento"; transfiere un byte desde la posición de memoria señalada por H y L hasta el destino señalado por D y E, y, además, decrementa BC. Incrementa automáticamente H y L, y D y E, de manera que todos los pares de registros quedan correctamente dispuestos para llevar a cabo la siguiente transferencia de bytes en el momento en que sea necesario.

LDIR significa "carga, incremento y repetición", y ejecuta LDI las veces necesarias para que el registro contador BC alcance el valor "0". Se utiliza para desplazar automáticamente un bloque continuo de datos desde un área de memoria a otra.

LDD y LDDR funcionan de la misma forma, con la diferencia de que los apuntadores de direcciones se *decrementan* en lugar de incrementarse; por tanto, la transferencia empieza por la dirección *más alta* del bloque en lugar de por la más baja. La figura 4.5 resume los resultados de las cuatro instrucciones

Figura 4.5
Grupo de transferencia de ... ques.

Hay instrucciones automáticas similares de comparación (CP) que se resumen en la figura 4.6.

			FUENTE: REG. INDIR. (HL)	
DESTINO	REG. INDIR.	(DE)	ED A0	'LDI' – cargar (DE) ← (HL) Inc HL & DE, Dec BC
			ED B0	'LDIR' – cargar (DE) ← (HL) Inc HL & DE, Dec BC, Repetir hasta que BC - 0.
			ED A8	'LDD' – cargar (DE) ← (HL) Dec HL & DE, Dec BC
			ED B8	'LDDR' – cargar (DE) ← (HL) Dec HL & DE, Dec BC, Repetir hasta que BC - 0.

Reg HL apunta a la fuente
Reg DE apunta al destino
Reg BC contador de bytes

Figura 4.5
Grupo de transferencia de bloques.

INSTRUCCIONES DE TRATAMIENTO DE DATOS

Aritméticas

Hay dos operaciones aritméticas básicas: suma y resta, que hemos empleado repetidas veces en el capítulo anterior. Hay dos tipos de suma, con y sin acarreo, codificadas como ADC y ADD, respectivamente. También hay dos instrucciones de resta, con y sin acarreo: SBC y SUB.

A este grupo pertenecen también las tres instrucciones especiales DAA, CPL y NEG. La instrucción de ajuste decimal del acumulador DAA sirve para realizar operaciones BCD, y normalmente se usa en todas las sumas y restas ejecutadas en ese código. CPL calcula el complemento a uno del acumulador, y NEG hace negativo el acumulador en el formato de su complemento (complemento a dos).

Todas las instrucciones anteriores operan sobre datos de ocho bits. Las operaciones de 16 bits son más limitadas. Como describe la figura 4.8, se dispone de ADD, ADC y SBC para registros específicos.

Por último, hay instrucciones de incremento y decremento que operan sobre todos los registros, tanto en el formato de 8 bits como en el de 16. Se resumen en la figura 4.7 (operaciones de 8 bits) y en la 4.8 (operaciones de 16 bits).

POSICION BUSCADA	
REG INDIR	
(HL)	
ED [illegible]	'CPI' Inc HL, Dec BC
ED 81	'CPIR', Inc HL, Dec BC Repetir hasta que BC = 0 o hasta coincidencia
ED A9	'CPD' Dec HL & BC
ED B9	'CPDR' Dec HL & BC Repetir hasta que BC = [illegible] o hasta coincidencia

HL señala la posición de memoria, que debe compararse con el contenido del acumulador
BC es el contador de bytes

Figura 4.6
Grupo de búsqueda de bloques.

FUENTE

	DIRECCIONAMIENTO DE REGISTROS							REG INDIR.	INDEXADO		[illegible]
	A	B	C	D	E	H	L	(HL)	(IX + d)	(IY + d)	[illegible]
ADD	87	80	81	82	83	84	85	86	DD 86 d	FD 86 d	[illegible]
SUMA con ACARREO 'ADC'	8F	88	89	8A	8B	8C	8D	8E	DD 8E d	FD [illegible] d	[illegible]
RESTA 'SUB'	97	90	91	92	93	94	95	96	DD 96 d	FD 96 d	[illegible]
RESTA con ACARREO 'SBC'	9F	98	99	9A	9B	9C	9D	9E	DD 9E d	FD 9E d	[illegible]
'AND'	A7	A0	A1	A2	A3	A4	A5	A6	DD A6 d	FD A6 d	[illegible]
'XOR'	AF	A8	A9	AA	AB	AC	AD	AE	DD AE d	FD AE d	[illegible]
'OR'	B7	B0	B1	B2	B3	B4	B5	B6	DD B6 d	FD B6 d	[illegible]
COMPARAR 'CP'	BF	B8	B9	BA	BB	BC	BD	BE	DD BE d	FD BE d	[illegible]
INCREMENTAR 'INC'	3C	04	0C	14	1C	24	2C	34	DD 34 d	FD 34 d	
DECREMENTAR 'DEC'	3D	05	0D	15	1D	25	2D	35	DD 35 d	FD 35 d	

Figura 4.7
Aritmética y lógica de 8 bits.

Figura 4.8
Aritmética y lógica de 16

Obsérvese que, en general, todas las operaciones aritméticas modifican algunas de las banderas; este aspecto se describe detalladamente en la exposición pormenorizada de las instrucciones contenida más adelante en este mismo capítulo. No obstante, hay que subrayar que las instrucciones INC y DEC que operan sobre pares de registros no modifican ninguna de las banderas; es un detalle importante que conviene tener en cuenta; supone que si se incrementa o se decrementa uno de los pares de registros hasta el valor "0", no se fijará a 1 el bit Z del registro de banderas F, de forma que es preciso verificar explícitamente en el programa si el valor del registro es "0".

También conviene señalar que las instrucciones ADC y SBC siempre afectan a todas las banderas. Esto no significa que necesariamente todas las banderas sean diferentes tras su ejecución, sino que pueden serlo.

DESTINO		FUENTE: BC	DE	HL	SP	IX	IY
'SUMA'	HL	09	19	29	39		
	IX	DD [illegible]	DD 19		DD 39	DD 29	
	IY	FD 09	FD 19		FD 39		FD 29
SUMA CON ACARREO Y ACTIVACION BANDERAS 'ADC'	HL	ED 4A	ED 5A	ED 6A	ED 7A		
RESTA CON ACARREO Y ACTIVACION BANDERAS 'SBC'	HL	ED 42	ED 52	ED 62	ED 72		
INCREMENTO 'INC'		03	13	23	33	DD 23	FD [illegible]
DECREMENTO 'DEC'		0B	1B	2B	3B	DD 2B	FD 2B

Figura 4.8
Aritmética y lógica de [illegible] bits.

Lógicas

Hay tres operaciones lógicas: AND, OR y XOR (exclusivo), y una instrucción de comparación CP. Todas ellas operan exclusivamente sobre datos de 8 bits. La tabla de la figura 4.7 recoge todas las posibilidades y códigos de operación de estas instrucciones, que estudiaremos ahora con cierto detalle.

AND

Cada operación lógica está caracterizada por una *tabla de verdad*, que expresa el valor lógico del resultado en función de las entradas. La tabla de verdad de AND es la siguiente:

0 AND 0 = 0
0 AND 1 = 0
1 AND 0 = 0
1 AND 1 = 1

ó

AND	0	1
0	0	0
1	0	1

La operación AND se caracteriza porque el resultado sólo es "1", si las dos entradas son "1"; en otras palabras, si una de las entradas es "0", el resultado siempre será "0". Esta característica se emplea para igualar a 0 un bit de una palabra, y se llama "enmascarar".

Una aplicación importante de la instrucción AND es eliminar o enmascarar uno o más bits específicos de una palabra. Supongamos que queremos igualar a 0 los cuatro bits derechos de una palabra; para ello utilizaremos el siguiente programa:

```
LD     A, PALABRA      PALABRA CONTIENE
                       "10101010"
AND    11110000B       "11110000" ES LA MAS-
                       CARA
```

Sea PALABRA igual a "10101010". Una vez ejecutado el programa, en el acumulador aparecerá el valor "10100000". "B" sirve para indicar un valor binario.

Ejercicio 4.1: *Redáctese un programa de tres líneas que iguale a 0 los bits 1 y 6 de PALABRA.*

Ejercicio 4.2: *¿Qué ocurriría si MASCARA = "11111111"?*

OR

Esta instrucción ejecuta la operación OR, caracterizada por la siguiente tabla de verdad:

0 OR 0 = 0
0 OR 1 = 1
1 OR 0 = 1
1 OR 1 = 1

ó

OR	0	1
0	0	1
1	1	1

Si uno de los operandos de la operación lógica OR es "1", el resultado es siempre "1". La aplicación obvia de la instrucción es igualar un bit a "1".

Igualemos a "1" los cuatro bits de la derecha de PALABRA:

```
LD      A, PALABRA
OR      00001111B
```

Supongamos que PALABRA contiene "10101010"; el valor final del acumulador será "10101111".

Ejercicio 4.3: *¿Cuál sería el resultado de la instrucción OR 10101111B?*

Ejercicio 4.4: *¿Cuál sería el resultado de aplicar la operación OR al número hexadecimal "FF"?*

XOR

XOR significa "OR exclusivo", que se diferencia del OR que acabamos de estudiar en que el resultado es "1" solamente si uno, y nada más que uno, de los operandos vale "1". En efecto, la operación OR normal produce como resultado "1" cuando los dos operandos valen "1", mientras que la operación exclusiva produce "0" en este mismo caso. La tabla de verdad es:

0 XOR 0 = 0
0 XOR 1 = 1
1 XOR 0 = 1
1 XOR 1 = 0

ó

XOR	0	1
0	0	1
1	1	0

Esta operación sirve para hacer comparaciones. Si dos palabras difieren en un bit cualquiera, el resultado del OR exclusivo será "1". Además, en el Z80 la instrucción sirve también para *complementar* una palabra, porque la instrucción de complemento sólo actúa sobre el acumulador. Para ello se ejecuta XOR con una palabra formada por unos. El programa es:

```
LD      A, PALABRA
XOR.    11111111B
```

Supongamos que PALABRA vale "10101010"; el valor final del registro será "01010101". Puede comprobarse que es el complemento del valor de partida.
XOR puede utilizarse como "conmutador de bit".

Ejercicio 4.5: *¿Cuál sería el resultado de aplicar XOR a un registro con el contenido hexadecimal "00"?*

OPERACIONES DE DESALINEAMIENTO (DESPLAZAMIENTO Y ROTACION)

Empecemos por diferenciar entre desplazamiento y rotación, operaciones ambas mostradas en la figura 4.9. El desplazamiento es el traslado del contenido de un registro en una posición de un bit hacia la izquierda o hacia la derecha. El bit que sale del registro pasa al acarreo C, y el que entra es 0, como ya se explicó en la sección anterior.

DESPLAZAMIENTO A LA IZQUIERDA

ACARREO

ROTACION A LA IZQUIERDA

ACARREO

Figura 4.9
Desplazamiento y rotación.

Hay una excepción, llamada *desplazamiento aritmético a la derecha*. Cuando se realizan operaciones con números negativos en formato complemento a 2, el bit de la izquierda corresponde al signo, y vale "1". Cuando se divide un número negativo por "2", desplazándolo a la derecha, debe mantenerse como negativo, lo que quiere decir que el bit de la izquierda debe seguir siendo "1". Todo esto lo ejecuta automáticamente la instrucción SRA de desplazamiento aritmético a la derecha. Con esta instrucción, el bit que entra por la izquierda es idéntico al de signo: es "0" si ése era su valor, y es "1" si valía "1". Los resultados se recogen en la figura 4.10, que resume todas las operaciones de desplazamiento y rotación posibles.

Rotación

La rotación se diferencia del desplazamiento en que el bit que se incorpora al registro procede bien del extremo opuesto

TIPO
DE
O
DES

Figura 4.10
Rotaciones y desplazamie

Figura 4.11
Rotación de 9 bits.

Figura 4.12
Rotación de 8 bits.

Fuente y destino

TIPO DE ROTACION O DESPLAZAMIENTO

A
07
0F
17
1F

Rotación circular a la izquierda
Rotación circular a la derecha
Rotación a la izquierda
Rotación a la derecha
Desplazamiento aritmético a la izquierda
Desplazamiento aritmético a la derecha
Desplazamiento lógico a la derecha
Rotación a la izquierda
Rotación a la derecha

Figura 4.10
Rotaciones y desplazamientos.

de ese mismo registro, bien del acarreo. El Z80 dispone de dos tipos de rotación, de 8 y de 9 bits.

La rotación de 9 bits se muestra en la figura 4.11. Si es hacia la derecha, los ocho bits del registro se desplazan uno hacia ese lado: el que sale por la derecha pasa al acarreo, como es habitual, cuyo valor anterior —antes de ser reemplazado por el que acaba de entrar— pasa al extremo izquierdo. En matemáticas se llama a esto rotación de 9 bits, porque supone el desplazamiento de los ocho bits del registro más el del acarreo. La rotación a la izquierda funciona exactamente igual, pero en sentido contrario.

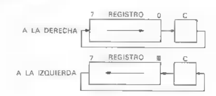


Figura 4.11
Rotación de 9 bits.

La rotación de 8 bits funciona igual. El bit 0 pasa a ocupar el lugar del bit 7 o viceversa, según el sentido de la operación. Además, el bit que sale del registro se introduce también en el acarreo. La secuencia queda mostrada en la figura 4.12.

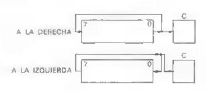


Figura 4.12
Rotación de ■ bits.

Instrucciones especiales para cifras

Hay dos instrucciones especiales de rotación de cifras q facilitan la aritmética BCD. El resultado es una rotación d cuatro bits entre dos cifras contenidas en la posición de memo ria señalada por los registros HL y la cifra que ocupa la mita inferior del acumulador. El funcionamiento se ilustra en l figura 4.13.

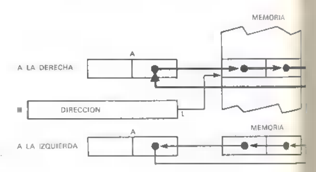


Figura 4.13
Instrucciones de rotación de cifras (rotación decimal).

MANIPULACION DE BITS

Ya hemos visto que las operaciones lógicas pueden utiliza para fijar o modificar bits o grupos de bits en el acumulado No obstante, resulta cómodo poder fijar o modificar cualqu bit de cualquier registro o de cualquier posición de la memori con una sola instrucción. Es una opción que exige un númer considerable de códigos de operación y que, por ello, falta en l mayor parte de los microprocesadores. Sin embargo, el Z8 cuenta con numerosas posibilidades de manipulación de bit que se resumen en la figura 4.14: la tabla recoge también l instrucciones de comprobación, que describiremos en la secció siguiente.

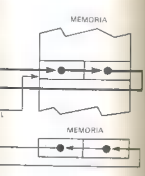
Hay dos instrucciones especiales para operar en la bander de acarreo: CCF (complementar bandera de acarreo) y SCF (poner la bandera de acarreo): las dos aparecen en la figur 4.15.

Nota: Se suele usar OR A o bien AND A para quitar l bandera de acarreo con una operación de un solo byte.

Figura 4.14
Instrucciones de manipul de bits.

as

ales de rotación de cifras que resultado es una rotación de nidas en la posición de memo- y la cifra que ocupa la mitad cionamiento se ilustra en la



iones lógicas pueden utilizarse pos de bits en el acumulador. er fijar o modificar cualquier quier posición de la memoria opción que exige un número ón y que, por ello, falta en la dores. Sin embargo, el Z80 les de manipulación de bits, la tabla recoge también las describiremos en la sección

es para operar en la bandera bandera de acarreo) y SCF s dos aparecen en la figura

bien AND A para quitar la ración de un solo byte.

	BIT	DIRECCIONAMIENTO DE REGISTROS							REG INDIR	INDEXADO	
		A	B	C	D	E	H	L	(HL)	(IX+d)	(IY+d)
VERIFICAR BIT	0	CB 47	CB 40	CB 41	CB 42	CB 43	CB 44	CB 45	CB 46	DD CB d 46	FD CB d 46
	1	CB 4F	CB 48	CB 49	CB 4A	CB 4B	CB 4C	CB 4D	CB 4E	DD CB d 4E	FD CB d 4E
	2	CB 57	CB 50	CB 51	CB 52	CB 53	CB 54	CB 55	CB 56	DD CB d 56	FD CB d 56
	3	CB 5F	CB 58	CB 59	CB 5A	CB 5B	CB 5C	CB 5D	CB 5E	DD CB d 5E	FD CB d 5E
	4	CB 67	CB 60	CB 61	CB 62	CB 63	CB 64	CB 65	CB 66	DD CB d 66	FD CB d 66
	5	CB 6F	CB 68	CB 69	CB 6A	CB 6B	CB 6C	CB 6D	CB 6E	DD CB d 6E	FD CB d 6E
	6	CB 77	CB 70	CB 71	CB 72	CB 73	CB 74	CB 75	CB 76	DD CB d 76	FD CB d 76
	7	CB 7F	CB 78	CB 79	CB 7A	CB 7B	CB 7C	CB 7D	CB 7E	DD CB d 7E	FD CB d 7E
PONER BIT A 0 RES	0	CB 87	CB 80	CB 81	CB 82	CB 83	CB 84	CB 85	CB 86	DD CB d 86	FD CB d 86
	1	CB 8F	CB 88	CB 89	CB 8A	CB 8B	CB 8C	CB 8D	CB 8E	DD CB d 8E	FD CB d 8E
	2	CB 97	CB 90	CB 91	CB 92	CB 93	CB 94	CB 95	CB 96	DD CB d 96	FD CB d 96
	3	CB 9F	CB 98	CB 99	CB 9A	CB 9B	CB 9C	CB 9D	CB 9E	DD CB d 9E	FD CB d 9E
	4	CB A7	CB A0	CB A1	CB A2	CB A3	CB A4	CB A5	CB A6	DD CB d A6	FD CB d A6
	5	CB AF	CB A8	CB A9	CB AA	CB AB	CB AC	CB AD	CB AE	DD CB d AE	FD CB d AE
	6	CB B7	CB B0	CB B1	CB B2	CB B3	CB B4	CB B5	CB B6	DD CB d B6	FD CB d B6
	7	CB BF	CB B8	CB B9	CB BA	CB BB	CB BC	CB BD	CB BE	DD CB d BE	FD CB d BE
PONER BIT A 1 SET	0	CB C7	CB C0	CB C1	CB C2	CB C3	CB C4	CB C5	CB C6	DD CB d C6	FD CB d C6
	1	CB CF	CB C8	CB C9	CB CA	CB CB	CB CC	CB CD	CB CE	DD CB d CE	FD CB d CE
	2	CB D7	CB D0	CB D1	CB D2	CB D3	CB D4	CB D5	CB D6	DD CB d D6	FD CB d D6
	3	CB DF	CB D8	CB D9	CB DA	CB DB	CB DC	CB DD	CB DE	DD CB d DE	FD CB d DE
	4	CB E7	CB E0	CB E1	CB E2	CB E3	CB E4	CB E5	CB E6	DD CB d E6	FD CB d E6
	5	CB EF	CB E8	CB E9	CB EA	CB EB	CB EC	CB ED	CB EE	DD CB d EE	FD CB d EE
	6	CB F7	CB F0	CB F1	CB F2	CB F3	CB F4	CB F5	CB F6	DD CB d F6	FD CB d F6
	7	CB FF	CB F8	CB F9	CB FA	CB FB	CB FC	CB FD	CB FE	DD CB d FE	FD CB d FE

Figura 4.14
Instrucciones de manipulación de bits

VERIFICACION Y SALTO

Dado que las operaciones de verificación dependen ■ gran medida de la manipulación del registro de estado, empezaremos por describir la función de cada una de las banderas. El contenido del registro puede estudiarse en la figura 4.16.

Ajuste decimal del acumulador, "DAA"	27
Complementar acumulador "CPL"	2F
Negativizar acumulador, "NEG" (complemento a dos)	ED 44
Complementar bandera de acarreo "CCF"	3F
Poner bandera de acarreo, "SCF"	37

Figura 4.15
Operaciones con AF de tipo general.

7	6	5	4	3	2	1	0
S	Z	-	H	-	P/V	N	C
(T)	(T)				(T)		(T)

Figura 4.16
Registro de estado.

C es el acarreo; N denota suma o resta; P/V, paridad o desbordamiento; H es el acarreo de la mitad; Z es cero, y S el signo. Los bits 3 y 5 ("–") del registro de estado no se utilizan. Las dos banderas H y N se emplean en aritmética BCD y no pueden verificarse. Las otras cuatro (C, P/V, Z y S) pueden verificarse en combinación con instrucciones condicionales de salto o llamada. La función de cada una de las banderas se describirá en los próximos párrafos.

Acarreo (C)

En casi todos los microprocesadores, y en el Z80 en particular, el bit de acarreo desempeña una función doble: en primer lugar, indica si una operación de suma o resta ha dado lugar a un acarreo; en segundo lugar, es el noveno bit de las instrucciones de desplazamiento y rotación. La reunión en un único bit de las dos funciones facilita algunas operaciones, como la multiplicación; esto ya debió quedar claro en la sección del capítulo anterior dedicada ■ dicha operación.

Al estudiar el empleo del bit de acarreo es importante recordar que todas las operaciones aritméticas lo igualan a 1 o a 0, dependiendo del resultado de las instrucciones. Lo mismo hacen todas las operaciones de desplazamiento y rotación, dependiendo el resultado del valor del bit que sale del registro.

Las instrucciones lógicas (AND, OR, XOR) siempre ponen a 0 el bit de acarreo, y pueden emplearse explícitamente para ello.

Las siguientes instrucciones afectan al bit de acarreo: ADD A,s; ADC A,s; SUB s; SBC A,s; CP s; NEG; AND s; OR s; XOR s; ADD DD,ss; ADC HL,ss; SBC HL,ss; RLA; RLCA; RRA; RRCA; RL m; RLC m; RR m; RRC m; SLA m; SRA m; SRL m; DAA; SCF; CCF.

Resta (N)

Normalmente, esta bandera no la usa el programador, sino el propio Z80 durante operaciones BCD. El lector recordará, sin duda, del capítulo anterior, que tras una suma o una resta BCD se ejecutaba una instrucción DAA (ajuste decimal del acumulador) para obtener resultados BCD válidos. Pero la operación de ajuste tras una suma es diferente que tras una resta, lo que quiere decir que la forma en que se ejecute la instrucción DAA depende del valor de la bandera N. Esta vale "0" tras una suma, y "1" tras una resta.

El símbolo "N" utilizado para esta bandera puede confundir a los programadores acostumbrados a otros microprocesadores, que quizá lo tomen erróneamente por el bit de signo. Es un bit interno de signo de operación.

Ponen N a "0" las instrucciones: ADD A,s; ADC A,s; AND s; OR s; XOR s; INC s; ADD DD,ss; ADC HL,ss; RLA; RLCA; RRA; RRCA; RL m; RLC m; RR m; RRC m; SLA m; SRA m; SRL m; RLD; RRD; SCF; CCF; IN r, (C); LDI; LDD; LDIR; LDDR; LD A, I; LD A, R; BIT b, s.

Ponen N a "1" las instrucciones: SUB s; SBC A,s; CP s; NEG; DEC m; SBC HL, ss; CPL; INI; IND; OUTI; OUTD; INIR; INDR; OTIR; OTDR; CPI; CPIR; CPD; CPDR.

Paridad/Desbordamiento (P/V)

La bandera de paridad/desbordamiento desempeña dos funciones diferentes. Mediante instrucciones específicas, la bandera se iguala a 1 o a 0 en función de la paridad del resultado, que se determina contando el número de unos del mismo; si este número es impar, el bit de paridad se hace igual a "0" (paridad impar); si es par, se iguala a "1" (paridad par). Donde más se utiliza la paridad [redacted] en los bloques de caracteres (por lo general, en formato ASCII). El bit de paridad es un bit adicional que se

añade al código de siete que representa el carácter, con el ■ garantizar la integridad de los datos almacenados en un dispo sitivo de memoria. De esta forma, ■, por ejemplo, uno de l bits del código que representa el carácter cambia accidentalme te a causa de un fallo del dispositivo de memoria (un disco RAM) o por un error de transmisión, cambiará también número total de unos del código, discrepancia que se detect verificando la bandera del bit de paridad. Dicha bandera emplea, en particular, en instrucciones lógicas y de rotació Como es natural, la misma bandera indicará la paridad de dato que se está leyendo durante las operaciones de entrada partir de un dispositivo de E/S.

El lector familiarizado con el 8080 deberá tener en cuent que en éste la bandera de paridad se usa exclusivamente com tal. En el Z80 desempeña algunas otras funciones adicionales debe, pues, manejarse con atención al pasar de un microproce sador al otro.

La segunda aplicación importante de esta bandera en el Z8 es el desbordamiento (no disponible en el 8080). La bandera de desbordamiento ya la estudiamos en el capítulo 1 al exponer l notación en complemento a dos. Detecta si, durante la suma la resta, cambia "accidentalmente" el signo del resultado debid al desbordamiento del mismo en el bit de signo (recuérdese qu en formato de 8 bits, el mayor positivo posible es + 127 y menor negativo − 128, siempre en complemento ■ dos).

Esta bandera realiza en el Z80 otras dos funciones que nad tienen que ver con las que acabamos de examinar.

Al ejecutar instrucciones de transferencia de bloques (LDD LDDR, LDI, LDIR) y de búsqueda (CPD, CPDR, CPI, CPIR) la bandera se emplea para comprobar si el registro contador ha alcanzado el valor "0". Con instrucciones que decrementa la bandera se pone ■ "0" ■ el par de registros del contador d byte BC es "0". Con instrucciones que incrementan se modific ■ BC − 1 = 0 al principio de la instrucción, es decir, si l instrucción reduce BC hasta "0".

Por último, al ejecutar las dos instrucciones especiale LD A,I y LD A,R, la bandera P/V refleja el valor del biestable d validación de interrupciones (IFF2); tal característica pued aprovecharse para conservar o verificar este valor.

Afectan a la bandera P: AND s; OR s; XOR s; RL ■; RLC m RR m; RRC m; SLA m; SRA m; SRL m; RLD; RRD; DAA IN r,(C).

Afectan a la bandera V: ADD A,s; ADC A,s; SUB s; SBC A CP s; NEG; INC s; DEC m; ADC HL,ss; SBC HL,ss.

También utilizan esta bandera LDIR; LDDR (puesta a "0" LDI; LDD; CPI; CPIR; CPD; CPDR.

Bandera de acarreo mitad (H)

Esta bandera revela el posible acarreo del bit 3 en el bit 4 durante una operación aritmética. En otras palabras, representa el acarreo del *nibble* de orden inferior en el de orden superior. Como es fácil deducir, se emplea, sobre todo, en operaciones BCD, y más concretamente, la utiliza el microprocesador para el ajuste decimal (DAA) necesario, con el fin de obtener como resultado un valor correcto.

La bandera se pone ■ 1 cada vez que se ejecuta una suma con acarreo del bit 3 al bit 4, y se vuelve al estado inicial si no hay tal acarreo. Y viceversa: en una resta se pone a 1 si hay acarreo del bit 4 al 3, y se vuelve al estado inicial en caso contrario.

Influyen en la bandera, la suma, la resta, el incremento, el decremento, las comparaciones y las operaciones lógicas.

Le afectan las instrucciones siguientes: ADD A,r; ADC A,s; SUB s; SBC A,s; CP s; NEG; AND s; OR s; XOR s; INC s; DEC m; RLA; RLCA; RRA; RRCA; RL m; RLC m; RR m; RRC m; SLA m; SR m; SRL m; RLD; RRD; DAA; CPL; SCF; IN r,(C); LDI; LLD; LDIR; LDDR; LD A; LD A,R; BIT b,r; CPI; CPIR; CPD; CPDR.

Obsérvese que el bit H se ve afectado aleatoriamente por las instrucciones de suma y resta de 16 bits y por las de entrada y salida de bloques.

Cero (Z)

La bandera Z se utiliza para verificar si es cero el valor de un byte que se ha calculado ■ que se está transfiriendo. También se usa en instrucciones de comparación, para señalar una coincidencia, y en algunas otras funciones de diversa índole.

Si se produce una operación con resultado cero o ■ se transfiere un dato y el byte vale cero, el bit Z se fija a "1"; en caso contrario, se lleva al valor inicial "0".

En instrucciones de comparación, Z vale "1", si el resultado de la comparación es afirmativo, y "0", en caso contrario.

En el Z80, esta bandera desempeña otras tres funciones: en conjunción con la instrucción BIT, se usa para indicar el valor del bit que se quiere comprobar; vale "1", si dicho bit es "0", y pasa al valor nulo en caso contrario.

Al utilizar instrucciones especiales de entrada y salida de bloques (INI, IND, OUTI, OUTD), la bandera Z vale "1", si $D - 1 = 0$, y "0" en caso contrario; también vale "1" ■ el contador de byte desminuye hasta "0" (INIR, INDR, OTIR, OTDR).

Por último, cuando se usan las instrucciones especiales IN r,(C), Z se pone a "1" para indicar que el byte de entrada vale "0".

El valor de Z depende, por tanto, de las siguientes instrucciones: ADD A,s; ADC A,s; SUB s; SBC A,s; CP s; NEG; AND s; OR s; XOR s; INC s; DEC m; ADC HL,ss; SBC HL,ss; RL m; RLC m; RR m; RRC m; SLA m; SRA m; SRL m; RLD; RRD; DAA; IN r, (C); INI; IND; OUTI; OUTD; INIR; INDR; OTIR; OTDR; CPI; CPIR; CPD; CPDR; LD A,I; LD A,R; BIT b,s; NEG.

Son instrucciones habituales que no modifican el bit Z: ADD DD,ss; RLA; RLCA; RRA; RRCA; CPL; SCF; CCF; LDI; LDD; LDIR; LDDR; INC DD; DEC DD.

Signo (S)

Esta bandera refleja el valor del bit más significatvo de un resultado o de un byte transferido (bit séptimo). En notación de complemento a 2, el bit más significativo representa el signo "0", si el número es positivo, y "1", si es negativo. El bit número siete se llama también, por ello, bit de signo.

En la mayor parte de los microprocesadores, el bit de signo representa un importante papel en la comunicación con los dispositivos de entrada/salida. Como casi ninguno dispone de instrucción BIT de verificación de bits específicos de los registros o de la memoria, el bit de signo suele ser el más fácil de comprobar. Si se desea examinar la situación de un dispositivo de entrada/salida, la lectura del registro de estado supone el condicionamiento automático del bit de signo, que toma el valor del bit siete del registro de estado y puede a continuación verificarse fácilmente por medio del programa. Esto explica por qué el registro de estado de la mayor parte de las pastillas de entrada/salida conectadas a sistemas microprocesadores tienen el indicador más importante (por lo general, dispuesto/no dispuesto) en la posición siete.

El Z80 dispone de una instrucción especial BIT. Sin embargo, para verificar una posición de memoria (que puede ser la dirección de un registro de estado de E/S) es preciso cargar primero la dirección en los registros IX, IY o HL. No existe ninguna instrucción para verificar directamente una dirección especificada de la memoria (es decir, que esta instrucción no tiene direccionamiento directo). Por tanto, el valor de una bandera de dispuesto correspondiente a un dispositivo de entrada/salida, y situada en la posición siete, es útil también en el Z80.

La bandera de signo también la utiliza la instrucción especial IN(C) para indicar el signo del dato que está leyendo.

El bit de signo se ve afectado por las instrucciones: ADD A,s; SUB s; SBC A,s; CP s; NEG; AND s; OR s; XOR s; INC s; DEC m; ADC HL,ss; SBC HL,ss; RL m; RLC m; RR m; RRC m; SLA m; SRA m; SRL m; RLD; RRD; DAA; IN r,(C); CPI; CPIR; CPD; CPDR; LD A,I; LD A,r; NEG; ADC A,s.

Resumen de banderas

Los bits de banderas se emplean para detectar automáticamente las situaciones especiales que se producen en el interior de la ALU del microprocesador. Como se detectan fácilmente por medio de instrucciones especiales, es sencillo emprender acciones específicas en respuesta a la situación detectada. Importa entender bien la función de todos los indicadores disponibles, porque la mayor parte de las decisiones internas del programa se adoptan en función de los mismos. Así, todos los saltos efectuados dentro del programa terminarán en una u otra posición, dependiendo de lo indicado por las banderas. La única excepción la constituyen las interrupciones, que describiremos en el capítulo de entrada y salida, y que pueden determinar el salto a una posición determinada siempre que en las patillas del Z80 se reciba una señal procedente del soporte físico.

De momento, basta con recordar la función principal de cada uno de los bits estudiados. Al escribir programas, el lector podrá consultar las descripciones detalladas de las instrucciones que figuran en este mismo capítulo, para verificar el efecto que cada una de ellas ejerce sobre las banderas. Normalmente pueden ignorarse casi todas, y quien todavía no esté familiarizado con ellas no debe, pues, dejarse intimidar por su aparente complejidad. Su funcionamiento irá quedando claro conforme vayamos estudiando nuevas aplicaciones.

La figura 4.17 contiene un resumen de las seis banderas y de la forma en que pasan a valer "1" ó "0" en virtud de las diferentes instrucciones.

Instrucciones de salto

En una bifurcación, el programa se desvía necesariamente a una dirección especificada. Es, pues, un punto en el que la ejecución secuencial se interrumpe para dar paso a un segmento distinto del programa. Los saltos pueden ser condicionales o incondicionales. En un salto incondicional se produce una bifurcación a una dirección determinada, con independencia de cualquier otra condición.

INSTRUCCIONES	C	Z	P/V	S	N	H	OBSERVACIONES
ADD A, s; ADC A, s	↕	↕	V	↕	0	↕	Suma de 8 bits o suma con acarreo
SUB s; SBC A, s; CP s; NEG	↕	↕	V	↕	1	↕	Resta de 8 bits, resta con acarreo, comparación y negación del acumulador
AND s	0	↕	P	↕	0	1	Operaciones lógicas
OR s, XOR s	0	↕	P	↕	0	0	Y activación diversas banderas
INC s	•	↕	V	↕	0	↕	Incremento de 8 bits
DEC m	•	↕	V	↕	1	↕	Decremento de 8 bits
ADD DD, ss	↕	•	•	•	0	X	Suma de 16 bits
ADC HL, ss	↕	↕	V	↕	0	X	Suma de 16 bits con acarreo
SBC HL, ss	↕	↕	V	↕	1	X	Resta de 16 bits con acarreo
RLA; RLCA, RRA, RRCA	↕	•	•	•	0	0	Rota el acumulador
RL m, RLC m, RR m, RRC m, SLA m, SRA m, SRL m	↕	↕	P	↕	0	0	Rotación y desplazamiento posición m
RLD, RRD	•	↕	P	↕	0	0	Rotación de dígito a izq. y der.
DAA	↕	↕	P	↕	•	↕	Ajuste decimal del acumulador
CPL	•	•	•	•	1	1	Complementar el acumulador
SCF	1	•	•	•	0	0	Poner acarreo
CCF	↕	•	•	•	0	X	Complementar acarreo
IN r, (C)	•	↕	P	↕	0	0	Entrada a registro indirecta
INI, IND, OUTI, OUTD	•	↕	X	X	1	X	Entrada y salida de bloques. Z = 0 si B ≠ 0; en caso contrario, Z = 1
INIR, INDR, OTIR, OTDR	•	1	X	X	1	X	
LDI, LDD	•	X	↕	X	0	0	Instrucciones de transferencia de bloques. P/V = 1 si BC ≠ 0; en caso contrario, P/V = 0
LDIR, LDDR	•	X	0	X	0	0	
CPI, CPIR, CPD, CPDR	•	↕	↕	X	1	X	Instrucciones de búsqueda de bloques. Z = 1 si A = (HL); en caso contrario, Z = 0. P/V = 1 si BC ≠ 0; en caso contrario, P/V = 0
LD A, I; LD A, R	•	↕	IFF	↕	0	0	El contenido del biestable de interrupción (IFF) se copia en la bandera P/V
BIT b, s	•	↕	X	X	0	1	El complemento del bit b de la posición s se copia en la bandera Z
NEG	↕	↕	V	↕	1	↕	Negación del acumulador

Por cortesía de Zilog, Inc.

Notación de la tabla.

SIMBOLO — **OPERACION**

C — Bandera acarreo unión. C = 1 si la operación produce acarreo del bit más significativo del operando o el resultado

Z — Bandera cero. Z = 1 si el resultado de la operación es cero

S — Bandera de signo. S = 1 si el BMS del resultado es uno

P/V — Bandera de paridad o desbordamiento. Paridad (P) y desbordamiento (V) comparten la misma bandera. Las operaciones lógicas afectan a esta bandera con la paridad del resultado, mientras que las aritméticas la afectan con el desbordamiento del resultado. Si P/V responde a la paridad, P/V = 1 si el resultado de la operación es par, y P/V = 0 si es impar. Si la bandera responde al desbordamiento, P/V = 1 si el resultado de la operación produce desbordamiento

H — Bandera de semiacarreo. H = 1 si las operaciones de suma o resta producen acarreo hacia o desde el bit 4 del acumulador

N — Bandera de suma/resta. N = 1 si la operación anterior fue una resta

Las banderas H y N se usan en combinación con la instrucción de ajuste decimal (DAA) para corregir el resultado en formato BCD empaquetado tras una suma o una resta con operandos en formato BDC empaquetado

↕ — La posición de la bandera depende del resultado de la operación.

• — La operación no modifica la bandera

0 — La operación pone la bandera a 0

1 — La operación pone la bandera a 1

X — No hay que preocuparse por la bandera.

V — La bandera P/V depende del desbordamiento del resultado de la operación

P — La bandera P/V depende de la paridad del resultado de la operación

r — Cualquiera de los registros de la UCP: A, B, C, D, E, H, L

s — Cualquier posición de 8 bits para todos los modos de direccionamiento compatibles con la instrucción de que se trate

ss — Cualquier posición de 16 bits para todos los modos de direccionamiento compatibles con la instrucción de que se trate

ii — Cualquiera de los dos registros de índice IX o IY

R — Contador de refresco

n — Valor de 8 bits dentro del intervalo 0–255

nn — Valor de 16 bits dentro del intervalo 0–65535

m — Cualquier posición de 8 bits para todos los modos de direccionamiento compatibles con la instrucción de que se trate

Figura 4.17
Resumen del funcionamiento de las banderas

En un salto condicional la secuencia de ejecución pasa a una dirección especificada sólo si se cumplen una o más condiciones. Es el tipo de instrucción que se emplea siempre que hay que tomar decisiones basadas en el valor de los datos o de los resultados calculados.

Para poder estudiar las instrucciones de salto condicionales es imprescindible entender la función del registro de estado, porque todas las decisiones de bifurcación se basan en las banderas. Dado que ya las estudiamos en la sección anterior, pasaremos ahora a analizar las instrucciones de salto de que dispone el Z80.

El microprocesdor cuenta con dos categorías de salto: el salto dentro del mismo programa y el salto a una subrutina (CALL) y la vuelta de la misma (RETURN). Tras cualquier instrucción de salto, el contador del programa PC se carga con una nueva dirección, a partir de la que se reanuda la ejecución del programa. Las posibilidades de las instrucciones de salto sólo pueden captarse plenamente en el contexto de los diversos modos de direccionamiento con que cuenta el microprocesador, por lo que dejaremos pendiente este aspecto de la discusión hasta el siguiente capítulo, que tratará de las técnicas de direccionamiento, y nos limitaremos aquí a estudiar otras características de tales instrucciones.

Como ya se ha dicho, los saltos pueden ser incondicionales (ramificación a una dirección de memoria especificada) o condicionales. En este caso, puede comprobarse una de las banderas Z, C, P/V o S para averiguar si vale "0" o "1".

Las abreviaturas correspondientes son:

Z = cero (Z = 1)
NZ = no cero (Z = 0)
C = acarreo (C = 1)
NC = no acarreo (C = 0)
PO = paridad impar
PE = paridad par
P = positivo (S = 0)
M = negativo (S = 1)

Además, el Z80 dispone de una instrucción combinada especial que decrementa el registro B y salta a una dirección de memoria especificada, siempre que no sea cero. Es una instrucción potente que se emplea para terminar bucles y que ya hemos tenido ocasión de poner a prueba en el capítulo anterior; su código simbólico es DJNZ.

También las instrucciones CALL y RET (retorno) pueden ser condicionales e incondicionales. Comprueban las mismas

banderas de las instrucciones de bifurcación que acabamos de ver.

Las de bifurcación condicional son instrucciones potentes; por lo general, faltan en otros microprocesadores de 8 bits. Mejoran la eficacia del programa, porque si no existiesen harían falta dos instrucciones para hacer lo mismo.

Hay dos instrucciones de retorno especiales reservadas a rutinas de interrupción, llamadas RETI y RETN, que estudiaremos en la sección de interrupciones del capítulo 6.

Los modos de direccionamiento y los códigos de operación de las bifurcaciones se encuentran en la figura 4.18.

			CONDICIÓN									
			IN-COND	ACARREO	SIN ACARREO	CERO	NO CERO	PARIDAD PAR	PARIDAD IMPAR	SIGNO NEG	SIGNO POS	[illegible]
SALTO 'JP'	INM EXT	nn	C3 n n	DA n n	D2 n n	CA n n	C2 n n	EA n n	E2 n n	FA n n	F2 n n	
SALTO 'JR'	RELATIVO	PC+e	18 e-2	38 e-2	30 e-2	28 e-2	20 e-2					
SALTO 'JP'	REG INDIR	(HL)	E9									
SALTO 'JP'	REG INDIR	(IX)	DD E9									
SALTO 'JP'	REG INDIR	(IY)	FD E9									
LLAMAR	INM EXT	nn	CD n n	DC n n	D4 n n	CC n n	C4 n n	EC n n	E4 n n	FC n n	F4 n n	
DECREMENTAR B SALTAR SI NO ES CERO DJNZ	RELATIVO	PC+e										[illegible]
RETORNO RET	REG INDIR	(SP) (SP+1)	C9	D8	D0	C8	C0	E8	E0	F8	F0	
RETORNO DE INT RETI	REG INDIR	(SP) (SP+1)	ED 4D									
RETORNO DE INT NO ENMASCARABLE RETN	REG INDIR	(SP) (SP+1)	ED 45									

Figura 4.18
Instrucciones de salto.

En el capítulo 5 se discutirán detenidamente los modos de direccionamiento. Al examinar la figura 4.18 se observa que muchos de ellos están restringidos. Así, el salto absoluto JP no puede comprobar cuatro banderas, pero sólo dos el salto JR.

Respecto a estas dos últimas instrucciones, conviene observar que, aunque JR suele emplearse más que JP, porque ocupa un byte menos y facilita la relocalización del programa, no son intercambiables (JR no puede verificar las banderas de paridad y signo).

La instrucción de *reinicio*, RST, es una bifurcación especial de un byte que permite saltar a cualquiera de las ocho direcciones iniciales del extremo inferior de la memoria (que son, en notación decimal, 0, 8, 16, 24, 32, 40, 48 y 56). Es una instruc-

Figura 4.19
Instrucción de reanudación.

ción potente, porque sólo ocupa un byte y proporciona una bifurcación rápida, por lo que se emplea, sobre todo, para responder a interrupciones. No obstante, el programador puede emplearla para cualquier otra cosa. La figura 4.19 recoge los códigos de operación de esta instrucción.

LLAMADA A LA DIRECCION		CO-DIGO OP	
	0000_H	C7	'RST 0'
	0008_H	CF	'RST 8'
	0010_H	D7	'RST 16'
	0018_H	DF	'RST 24'
	0020_H	E7	'RST 32'
	0028_H	EF	'RST 40'
	0030_H	F7	'RST 48'
	0038_H	FF	'RST 56'

H denota un número hexadecimal

Figura 4.19
Instrucción de reanudación.

Instrucciones de entrada/salida

Las técnicas de entrada y salida se describirán pormenorizadamente en el capítulo 6. Baste decir aquí que los dispositivos de E/S pueden direccionarse de dos modos: como posiciones de memoria, utilizando cualquiera de las instrucciones que ya se han descrito para ello, o mediante instrucciones específicas de entrada/salida. Las instrucciones normales de direccionamiento de la memoria requieren tres bytes, uno para el código de operación y dos para la dirección, y otros tantos accesos a la memoria; son, por tanto, lentas. Las instrucciones de E/S especializadas son más breves y rápidas, pero tienen dos inconvenientes.

En primer lugar, "desperdician" varios de los preciosos y escasos códigos de operación disponibles (en efecto, en un microprocesador suelen utilizarse sólo 8 bits para formar todos

los códigos de operación necesarios). En segundo lugar, exige la emisión de una o más señales especializadas de entrada/salida; por tanto, también "desperdician" una o más de las pocas patillas de que dispone el microprocesador, casi siempre limitadas a 40. Debido a estos inconvenientes, los procesadores suelen carecer de instrucciones específicas de entrada/salida; sin embargo, dispone de ellas el 8080 (el primer microprocesdor de 8 bits de tipo general potente) y el Z80, que, como sabemos, es compatible con el 8080.

La ventaja de las instrucciones de entrada/salida es que son más rápidas, porque sólo ocupan dos bytes; no obstante, puede conseguirse un resultado similar con un modo de direccionamiento especial llamado direccionamiento de "página 0", que limita la dirección a un campo de 8 bits. Es la solución habitualmente elegida en otros microprocesadores.

Las dos instrucciones básicas de entrada/salida son IN y OUT, que transfieren el contenido de las posiciones de E/S especificadas a cualquiera de los registros de trabajo, o viceversa. Normalmente ocupan dos bytes: el primero, reservado para el código de operación, y el segundo, para la parte inferior de la dirección. El acumulador se emplea para entregar la parte superior de la misma, lo que permite seleccionar uno de los dispositivos de 64K. No obstante, esto exige cargar cada vez el acumulador con el contenido adecuado, lo que puede reducir la velocidad de ejecución.

Además, el Z80 dispone de un modo de registro indirecto más cuatro instrucciones especializadas de transferencia de bloques para entrada y salida.

En modo de *entrada a registro*, cuyo formato es IN r,(C), el par de registros B y C se usa como apuntador del dispositivo de E/S. El contenido de B se deja en la parte superior del *bus* de direcciones, y a continuación se carga el contenido del dispositivo de E/S especificado en el registro designado por r.

Lo mismo cabe decir de la instrucción OUT.

Las cuatro instrucciones de transferencia de bloques a la entrada son: INI, INIR (INI repetida), IND ■ INDR (IND repetida). Las correspondientes para la salida son: OUTI, OTIR, OUTD y OTDR.

En esta transferencia automática de bloques, el par de registros H y L se utiliza como apuntador de destino, y el registro C, como selector de dispositivo de E/S (uno entre 256 dispositivos). En las instrucciones de salida, H y L apuntan a la fuente. El registro B se usa como contador, y puede incrementarse o decrementarse; las instrucciones de entrada correspondientes son INI para incrementar e IND para decrementar.

INI es una instrucción de transferencia automática de un

Figura 4.20
Instrucciones de salida.

byte. El registro C selecciona el dispositivo de entrada. En este dispositivo se lee un byte y se transfiere a la dirección de memoria apuntada por H y L, que a continuación se incrementan en 1; por su parte, el contador B se decrementa en 1.

INIR es la misma instrucción automatizada: se ejecuta una y otra vez hasta que el contador se decrementa hasta "0". De esta forma pueden transferirse automáticamente hasta 256 bytes. Obsérvese que, para conseguir la transferencia total de 256 bytes exactamente, el registro B debe fijarse al valor "0" antes de ejecutar esta instrucción.

Los códigos de operación de las instrucciones de entrada y salida se resumen en las figuras 4.20 y 4.21.

FUENTE

			REGISTRO							REG INDIR
			A	B	C	D	E	H	L	(HL)
OUT	INMED.	(n)	D3 n							
	REG INDIR	(C)	ED 79	ED 41	ED 49	ED 51	ED 59	ED 61	ED 69	
SALIDA OUTI Inc HL, Dec. B	REG. INDIR	(C)								ED A3
SALIDA OTIR, Inc HL Dec. B, REPITE SI B ≠ 0	REG. INDIR	(C)								ED B3
SALIDA OUTD Dec HL & B	REG INDIR	(C)								ED AB
SALIDA OTDR, Dec HL y B REPITE SI B ≠ 0	REG INDIR	(C)								ED BB

ORDENES DE SALIDA DE BLOQUES

DIRECCION DE LA PUERTA DE DESTINO

Figura 4.20
Instrucciones de salida.

INSTRUCCIONES DE CONTROL

Estas instrucciones modifican la situación operativa de la CPU o manipulan la información de su estado interno. Son siete.

NOP es una instrucción de no operación, que mantiene el procesador inactivo durante un ciclo. Acostumbra a utilizarse para introducir un retraso deliberado (4 estados = 2 microsegundos con un reloj de 2 MHz) o para cubrir los huecos formados en un programa durante la fase de corrección. Para facilitar ésta, el código de operación de NOP suele estar formado íntegramente por ceros, porque en el momento de la ejecución la memoria suele limpiarse, es decir, suele reducirse a ceros; la ejecución de NOP no provoca ningún daño y no interrumpe la ejecución del programa.



Figura 4.21
Instrucciones de entrada.

HALT se emplea combinada con interrupciones ■ con el reinicio. Lo que hace es interrumpir el funcionamiento de la CPU; ésta reanudará el funcionamiento en cuanto reciba una señal de interrupción o de reinicio. En este modo, la CPU ejecuta continuamente instrucciones NOP. Durante la fase de corrección suele colocarse un alto al final del programa, porque habitualmente el programa principal no tiene que hacer ninguna otra cosa. Hecho esto, es necesario volver a ponerlo en funcionamiento explícitamente.

Hay dos instrucciones especiales para invalidar y validar la bandera interna de interrupciones: EI y DI. Describiremos las interrupciones en el capítulo 6. La bandera de interrupciones sirve para autorizar o desautorizar la interrupción de un programa. Para evitar interrupciones durante una porción específica del programa, puede invalidarse mediante esta instrucción el biestable o bandera de interrupciones, como veremos en el capítulo 6. La figura 4.22 recoge estas instrucciones.

Por último, el Z80 cuenta con tres modos de interrupción (frente ■ sólo uno en el 8080). El modo 0 corresponde al único del 8080; el 1 es una llamada a la posición 038H, y el 2 es una llamada indirecta que utiliza el contenido del registro especial I más 8 bits que proporciona el dispositivo interruptor como puntero de la posición de memoria cuyo contenido corresponde a la dirección de la rutina de interrupción. Explicaremos estos modos en el capítulo 6.

Figura 4.22
Instrucciones de control de la CPU.

Resumen

'NOP'	00	
'HALT'	76	
INVALIDA INT. "(DI)"	F3	
VALIDA INT "(EI)"	FB	
MODO DE INT. 0 "IM0"	ED 46	MODO 8080A
MODO DE INT. 1 "IM1"	ED 56	LLAMADA A LA POSICION 0038_H
MODO DE INT. 2 "IM2"	ED 5E	LLAMADA INDIRECTA USANDO EL REGISTRO I Y 8 BITS DEL DISPOSITIVO INTERRUPTOR COMO PUNTERO

Figura 4.22
Instrucciones de control de la CPU

El Z80 puede recibir dos tipos de interrupciones, que llegan por las patillas IRQ y NMI, y que también analizaremos en el capítulo 6.

Resumen

Ya hemos visto las cinco categorías de instrucciones de que dispone el Z80. La sección siguiente recoge los detalles particulares de cada una de ellas. Para empezar a programar no es preciso entender la función de todas las instrucciones, sino que basta con conocer unas pocas esenciales. No obstante, quien desee escribir buenos programas sí tendrá que estudiarlas y conocerlas todas. Como es natural, al principio la eficacia tiene poca importancia, y por eso pueden ignorarse la mayoría de las instrucciones.

Todavía no hemos descrito un aspecto muy importante: las técnicas de direccionamiento del Z80, que facilitan la recuperación de datos archivados en el espacio de la memoria. Estudiaremos dichas técnicas en el capítulo siguiente.

INSTRUCCIONES DEL Z80 DESCRIPCION INDIVIDUAL

ABREVIATURAS

BANDERA	ON	OFF
Acarreo	C (acarreo)	NC (no acarreo)
Signo	M (menos)	P (más)
Cero	Z (cero)	NZ (no cero)
Paridad	PE (par)	PO (impar)

● cambia funcionalmente según la operación
0 bandera a cero
1 bandera a uno
? bandera determinada aleatoriamente por la operación
x caso especial: véase la nota que figura en la misma página

Los bits de las posiciones 3 y 5 son siempre aleatorios

ADC A, s

Función:

Formato:

(H

(IX +

(IY +

Descripción:

Flujo de datos:

ADC A, s

Suma con acarreo del acumulador y el operando indicado.

Función: A ← A + s + C

Formato: s puede ser r, n, (HL), (IX + d) o (IY + d)

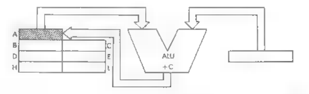
r	1 0 0 0 1 — r —	
n	1 1 0 0 1 1 1 0	byte 1: CE
	— n —	byte 2: dato inmediato
(HL)	1 0 0 0 1 1 1 0	8E
(IX + d)	1 1 0 1 1 1 0 1	byte 1: DD
	1 0 0 0 1 1 1 0	byte 2: 8E
	— d —	byte 3: valor del desplazamiento
(IY + d)	1 1 1 1 1 1 0 1	byte 1: FD
	1 0 0 0 1 1 1 0	byte 2: 8E
	— d —	byte 3: valor del desplazamiento

r puede ser:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Descripción: El operando s y la bandera de acarreo C del registro de estado se suman al acumulador, y el resultado se almacena en éste: s se define en la descripción de instrucciones ADD similares.

Flujo de datos:



Tiempo:

s	Ciclos M	Estados T	µseg @ 2 MHz
r	1	4	2
n	2	7	3.5
(HL)	2	7	3.5
(IX + d)	5	19	9.5
(IY + d)	5	19	9.5

Direccionamiento: r: implícito; n: inmediato; (HL): indirecto; (IX + d), (IY + d): indexado.

Códigos byte: ADC A,r r:

A	B	C	D	E	H	L
[illegible]	88	89	8A	8B	8C	8D

Banderas:

S	Z		H		P/V	N	C
●	●		●		●	0	●

Ejemplo: ADC A, 1A

Antes:



Después:



CE
1A
CODIGO OBJETO

ADC HL, ss

Función:

Formato:

Descripción:

Flujo de datos:

Tiempo:

Direccionamiento:

Códigos byte:

Banderas:

ADC HL, ss

Suma con acarreo de HL y el par de registros ss.

Función:

HL ← HL + ss + C

Formato:

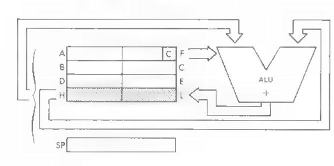
1	1	1	0	1	1	0	1	byte 1: ED
0	1	s	s	1	0	1	0	byte 2

Descripción:

El contenido del par de registros HL se suma al del par especificado y a continuación se suma el contenido de la bandera de acarreo. El resultado final se almacena en HL. El par ss puede ser:

BC – 00 HL – 10
DE – 01 SP – 11

Flujo de datos:



Tiempo:

4 ciclos M: 15 estados T: 7.5 μseg @ 2 MHz.

Direccionamiento:

Implícito.

Códigos byte:

ss:

	BC	DE	HL	SP
ED–	4A	5A	6A	7A

Banderas:



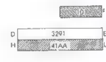
H se pone a 1 si hay arrastre del bit 11.

Ejemplo: ADC HL, DE

Antes: Después:







ADD A, (HL)

Función:

Formato:

Descripción:

Flujo de datos:

Tiempo:

Direccionamiento:

Banderas:

Ejemplo:





ADD A, (HL)

Suma el acumulador con la posición de memoria (HL) direccionada indirectamente.

Función: A ← A + (HL)

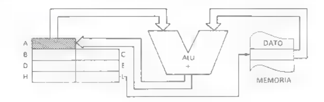
Formato:

1	0	0	0	0	1	1	0

86

Descripción: El contenido del acumulador se suma al de la posición de memoria direccionada por el par de registros HL. El resultado se almacena en el acumulador.

Flujo de datos:



Tiempo: 2 ciclos M; 7 estados T; 3.5 μseg @ 2 MHz

Direccionamiento: Indirecto.

Banderas:

S	Z		H		P/V	N	C
●	●		●		●	0	●

Ejemplo: ADD A,(HL)

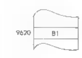
Antes:



Después:









ADD A, (IX+d)

Suma el acumulador con la posición de memoria indexada (IX + d).

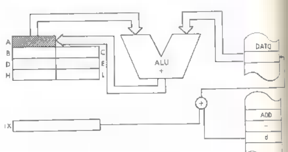
Función: A ← A + (IX + d)

Formato:

1	1	0	1	1	1	0	1	byte 1: DD
1	0	0	0	0	1	1	0	byte 2: 86
←			d				→	byte 3: valor del desplazamiento

Descripción: El contenido del acumulador se suma al de la posición de memoria direccionada por el contenido del registro IX más el valor de desplazamiento inmediato. El resultado se almacena en el acumulador.

Flujo de datos:



Tiempo: 5 ciclos M; 19 estados T: 9.5 μseg @ 2 MHz.

Direccionamiento: Indexado.

Banderas:

S	Z		H		P/V	N	C
●	●		●		●	0	●

Ejemplo:

DD
86
03
CODIGO OBJETO

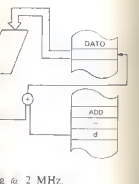
ción de memoria indexada

DD

86

valor del desplazamiento

suma al de la posición de
enido del registro IX más el
El resultado se almacena en



g @ 2 MHz.

Ejemplo:

ADD A,(IX + 3)

Código objeto
DD
86
03

CODIGO OBJETO

Antes:

A: 11

IX: 0B61

0B61	04
0B62	B2
0B63	36
0B64	91

Después:

A: 47

IX: 0B61

0B61	04
0B62	B2
0B63	36
0B64	91

ADD A, (IY + d)

Suma el acumulador con la posición de memoria indexada (IY + d).

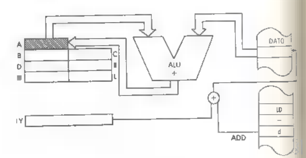
Función: A ← A + (IY + d)

Formato:

1	1	1	1	1	1	0	1	byte 1: FD
1	0	0	0	0	1	1	0	byte 2: 86
d								byte 3: valor del desplazamiento

Descripción: El contenido del acumulador se suma al contenido de la posición de memoria direccionada por el contenido del registro IY más el valor de desplazamiento dado. El resultado se almacena en el acumulador.

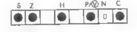
Flujo de datos:



Tiempo: 5 ciclos M; 19 estados T; 9.5 μseg @ 2 MHz

Direccionamiento: Indexado.

Banderas:



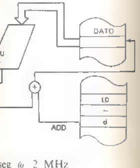
sición de memoria indexada

FD

86

valor del desplazamiento

uma al contenido de la posi-
r el contenido del registro IY
ado. El resultado se almacena

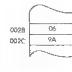


seg @ 2 MHz

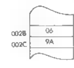
Ejemplo: ADD A,(IY + 1)



Antes:



Después:



ADD A, n

Suma el acumulador con el dato inmediato n.

Función: A ← A + n

Formato:

1	1	0	0	0	1	1	0

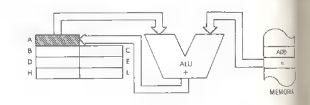
byte 1: C6

← n →

byte 2: dato inmediato

Descripción: El contenido del acumulador se suma al de la posición de memoria inmediata al código de operación. El resultado se almacena en el acumulador.

Flujo de datos:



Tiempo: 2 ciclos M; 7 estados T: 3.5 μseg @ 2 MHz.

Direccionamiento: Inmediato.

Banderas

S	Z		H		P/Ⓥ	N	C
●	●		●		●	0	●

Ejemplo: ADD A, E2

Antes: A 43

Después: A 25

C6
E2

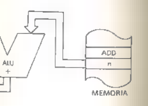
OBJETO CODIGO

o inmediato n.

: C6

: dato inmediato

e suma al de la posición de
e operación. El resultado se



eg @ 2 MHz.

espués:

ADD A, r

Suma el acumulador con el registro r.

Función: A ← A + r

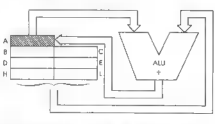
Formato:

1	0	0	0	0	← r →

Descripción: El contenido del acumulador se suma al del registro especificado. El resultado se coloca en el acumulador; r puede ser:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Flujo de datos:



Tiempo: 1 ciclo M; 4 estados T; 2 μseg @ 2 MHz.

Direccionamiento: Implícito.

Códigos byte:

S	Z		H		P/V	N	C
●	●		●		●	0	●

Banderas:

A	B	C	D	E	H	L
■	80	■	■	83	■	■

Ejemplo: ADD A, B

Antes:		Después:	
A	3D	A	3F
B	02	B	02

CODIGO OBJETO: ■

ADD HL, ss

Suma HL y el par de registros ss.

Función: HL ← HL + ss

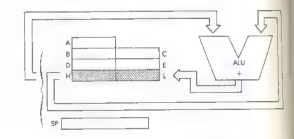
Formato:

0	0	s	s	1	0	0	1

Descripción: El contenido del par especificado se suma al del par HL y el resultado se almacena en HL; ss puede ser:

BC — 00 HL — 10
DE — 01 SP — 11

Flujo de datos:



Tiempo: 3 ciclos M; 11 estados T; 5.5 μseg @ MHz.

Direccionamiento: Implícito.

Códigos byte: ss:

BC	DE	HL	SP
09	19	29	39

Banderas:

S	Z		H		P/V	N	C
			?			0	●

El acarreo del bit 15 pone C a 1; en caso contrario, vale 0.
El acarreo del bit 11 pone H a 1.

Ejemplo: ADD HL, HL

Antes: H 0681 L

Después: H 0D02 L

29
CODIGO OBJETO

ADD IX, rr

Función:

Formato:

Descripción:

Flujo de datos:

Tiempo:

Direccionamiento:

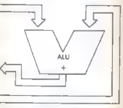
Códigos byte:

Banderas:

ss.

se suma al del par HL, y el
puede ser:

HL – 10
SP – 11



eg @ MHz.

l; en caso contrario, vale 0.
1.

spués:



ADD IX, rr

Suma IX con el par de registros rr.

Función: IX ← IX + rr

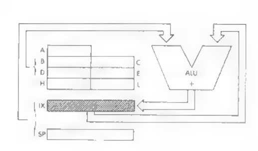
Formato:

1	1	0	1	1	1	0	1	byte 1: DD
0	0	r	r	1	0	0	1	byte 2

Descripción: El contenido del registro IX se suma al del par de registros especificado, y el resultado se almacena en IX; rr puede ser:

BC – 00   IX – 10
DE – 01   SP – 11

Flujo de datos:



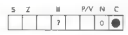
Tiempo: 4 ciclos M; 15 estados T; 7.5 μseg @ 2 MHz.

Direccionamiento: Implícito.

Códigos byte:

rr:	BC	DE	IX	SP
DD-	09	19	29	39

Banderas:



S	Z		H		P/V	N	C
			?			0	●

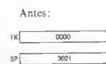
H se pone a 1 si hay acarreo desde el bit 11.
C se pone a 1 si hay acarreo desde el bit 15.



Ejemplo: ADD IX,SP



Antes:



Después:



ADD IY, rr

Función:

Formato:

Descripción:

Flujo de datos:

Tiempo:

Direccionamiento:

Códigos byte:

Banderas:

spués:

3021

3021

ADD IY, rr

Suma IY y el par de registros rr.

Función: IY ← IY + rr

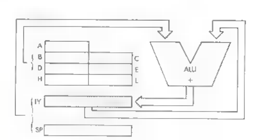
Formato:

1	1	1	1	1	1	0	1	byte 1: FD
0	0	r	r	1	0	0	1	byte 2

Descripción: El contenido del registro IY se suma al del par especificado, y el resultado se almacena en IY: rr puede ser:

BC – 00 IY – 10
DE – 01 SP – 11

Flujo de datos:



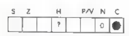
Tiempo: 4 ciclos M; 15 estados T: 7.5 μseg @ 2 MHz.

Direccionamiento: Implícito.

Códigos byte: rr:

	BC	DE	IY	SP
FD-	09	19	29	39

Banderas:

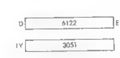


H vale 1 si hay acarreo del bit 11.
C vale 1 si hay acarreo del bit 15.

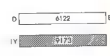
Ejemplo: ADD IY, DE



Antes:



Después:



AND ■

Función:

Formato:

(HI

(IX +

(IY +

Descripción:

Flujo de datos:

AND s

"Y" lógica del acumulador y el operando s.

Función: A ← A ∧ s

Formato: s puede ser: r, n, (HL), (IX + d) o (IY + d)

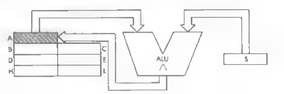
r	1 0 1 0 0 ← r →	
n	1 1 1 0 0 1 1 0	byte 1: E6
	← n →	byte 2: dato inmediato
(HL)	1 0 1 0 0 1 1 0	A6
(IX + d)	1 1 0 1 1 1 0 1	byte 1: DD
	1 0 1 0 0 1 1 0	byte 2: A6
	← d →	byte 3: valor del desplazamiento
(IY + d)	1 1 1 1 1 1 0 1	byte 1: FD
	1 0 1 0 0 1 1 0	byte 2: A6
	← d →	byte 3: valor del desplazamiento

r puede ser:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Descripción: El acumulador y el operando especificado se someten a la operación lógica "Y" (AND), y el resultado se almacena en el acumulador; s se define en la descripción de instrucciones ADD similares.

Flujo de datos:



Tiempo:

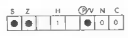
s	Ciclos M	Estados T	µseg @ 2 MH
r	1	4	2
n	2	7	3.5
(HL)	2	7	3.5
(IX + d)	5	19	9.5
(IY + d)	5	19	9.5

Direccionamiento: r: implícito; n: inmediato; (HL): indirecto; (IX + d), (IY + indexado.

Códigos byte: AND r

r	A	B	C	D	E	H	L
	A7	A0	A1	A2	A3	A4	A5

Banderas:



Ejemplo: AND 4B

Antes: A 36

Después: A 02



BIT b, (HL)

Función:

Formato:

Descripción:

Flujo de datos:

Tiempo:

Direccionamiento:

Banderas:

Códigos byte:

Estados T	μseg @ 2 MHz
4	2
7	3.5
7	3.5
19	9.5
19	9.5

ndirecto: (IX + d), (IY + d):

spués:

BIT b, (HL)

Verifica el bit b de la posición de memoria (HL) direccionada indirectamente.

Función: $Z \leftarrow \overline{(HL)_b}$

Formato:

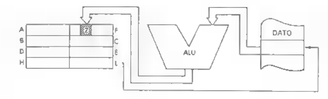
1	1	0	0	1	0	1	1	byte 1: CB
0	1	←	b	→	1	1	0	byte 2

Descripción:

Se verifica el bit especificado de la posición de memoria direccionada por el contenido del registro HL y se activa la bandera Z en función del resultado; b puede ser:

0 – 000	4 – 100
1 – 001	5 – 101
2 – 010	6 – 110
3 – 011	7 – 111

Flujo de datos:



Tiempo: 3 ciclos M: 12 estados T: 6 μseg @ 2 MHz.

Direccionamiento: Indirecto.

Banderas:

S	Z		H		P/V	N	C
?	●		1		?	0	

Códigos byte:



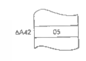
b:	0	1	2	3	4	5	6	7
CB-	46	4E	56	5E	66	6E	76	7E

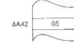
Ejemplo: BIT 3,(HL)

Antes: | Después:



Antes		Después	
F	00	F	54
HL	6A42	HL	6A42





CB
5E
CODIGO OBJETO

BIT b, (IX + d)

Función:

Formato:

Descripción:

Flujo de datos:

Tiempo:

Direccionamiento:

BIT b, (IX + d)

Verifica el bit b de la posición de memoria indexada (IX + d).

Función: $Z \leftarrow \overline{(IX + d)_b}$

Formato:

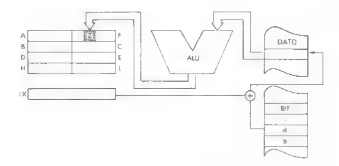
1 1 0 1 1 1 0 1	byte 1: DD
1 1 0 0 1 0 1 1	byte 2: CB
← d →	byte 3: valor del desplazamiento
0 1 ← b → 1 1 0	byte 4:

Descripción:

El bit especificado de la posición de memoria direccionada por el contenido del registro IX más el valor del desplazamiento dado se verifica, y se activa la bandera Z en función del resultado; b puede ser:

0 – 000	4 – 100
1 – 001	5 – 101
2 – 010	6 – 110
3 – 011	7 – 111

Flujo de datos:



Tiempo: 5 ciclos M; 20 estados T; 10 μseg @ 2 MHz.

Direccionamiento: Indexado.

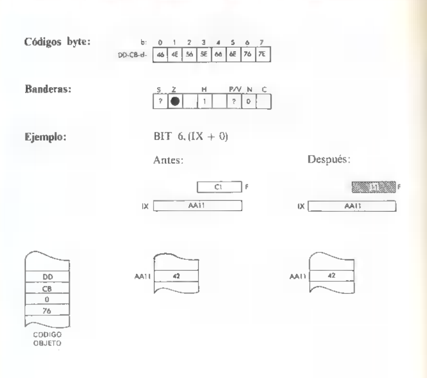
Códigos byte:

b:	0	1	2	3	4	5	6	7
DD-CB-d-	46	4E	56	5E	66	6E	76	7E

Banderas:

S	Z		H		P/V	N	C
?	●		1		?	0	

Ejemplo: BIT 6,(IX + 0)



BIT b,(IY + d

Función:

Formato:

Descripción:

Flujo de datos:

Tiempo:

Direccionamiento

BIT b, (IY + d)

Verifica el bit b de la posición de memoria indexada (IY + d).

Función: $Z \leftarrow \overline{(IY + d)_b}$

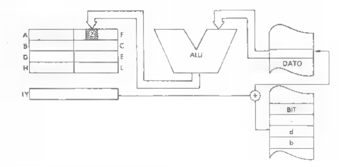
Formato:

1 1 1 1 1 1 0 1	byte 1: FD
1 1 0 0 1 0 1 1	byte 2: CB
←—— d ——→	byte 3: valor del desplazamiento
0 1 ← b → 1 1 0	bute 4

Descripción: Se verifica el bit especificado de la posición de memoria direccionada por el contenido del registro IY más el valor del desplazamiento dado y se activa, en consecuencia, la bandera Z; b puede ser:

0 – 000	4 – 100
1 – 001	5 – 101
2 – 010	6 – 110
3 – 011	7 – 111

Flujo de datos:



Tiempo: 5 ciclos M; 20 estados T; 10 μseg @ 2 MHz.

Direccionamiento: Indexado.

Códigos byte:

	0	1	2	3	4	5	6	7
FD-CB-d-	46	4E	56	5E	66	6E	76	7E

Banderas:

S	Z		H		P/V	N	C
?	●		1		?	0	

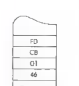
Ejemplo: BIT 0,(IY + 1)

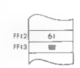
Antes:

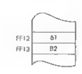


Después:









BIT b, r

Función:

Formato:

Descripción:

Flujo de datos:

Tiempo:

Direccionamiento:

Códigos byte: CB-

BIT b, r

Verifica el bit b del registro r.

Función: $Z \leftarrow \overline{r_b}$

Formato:

1	1	0	0	1	0	1	1	byte 1: CB
0	1	← b →			← r →			byte 2

Descripción: Verifica el bit especificado del registro dado y activa la bandera cero en función del resultado: b y r pueden ser:

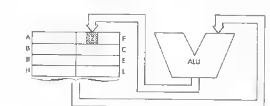
b:

0 – 000	4 – 100
1 – 001	5 – 101
2 – 010	6 – 110
3 – 011	7 – 111

r:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

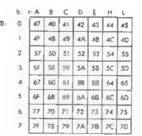
Flujo de datos:



Tiempo: 2 ciclos M; 8 estados T: 4 μseg @ 2 MHz.

Direccionamiento: Implícito.

Códigos byte:



	b: r:	A	B	C	D	E	H	L
CB-	0	47	40	41	42	43	44	45
	1	4F	4B	49	4A	4B	4C	4D
	2	57	50	51	52	53	54	55
	3	5F	58	59	5A	5B	5C	5D
	4	67	60	61	[illegible]	[illegible]	64	65
	5	6F	68	69	6A	6B	6C	6D
	6	77	70	71	72	73	74	75
	7	7F	78	79	7A	7B	7C	7D

Banderas:

S	Z		H		P/V	N	C
?	●		1		?	0	

Ejemplo:

BIT 4, B

Antes: B 61, 01 F

Después: B 61, 55 F

CB
60
CODIGO OBJETO

CALL cc, pq

Función:

Formato:

Descripción:

Flujo de datos:

CALL cc, pq

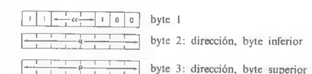
Llamada condicional a subrutina.

Función:

Si cc cierto: $(SP - 1) \leftarrow PC_{sup}$; $(SP - 2) \leftarrow PC_{inf}$; $SP \leftarrow SP - 2$; $PC \leftarrow pq$

Si cc falso: $PC \leftarrow PC + 3$

Formato:



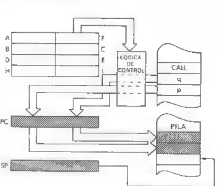
Descripción:

Si se satisface la condición, el contenido del contador del programa se empuja en la pila tal como se describe en las instrucciones PUSH. A continuación, el contenido de la posición de memoria inmediatamente siguiente al código de operación se carga en la parte inferior del PC y el contenido de la posición de memoria siguiente se carga en la mitad superior del PC. La siguiente instrucción se tomará de esta nueva dirección. Si la condición no se satisface, se ignora la dirección pq y se ejecuta la instrucción siguiente; cc puede ser:

NZ – 000	PO – 100
Z – 001	PE – 101
NC – 010	P – 110
C – 011	M – 111

Al final de la subrutina llamada puede usarse una instrucción RET para restablecer el PC.

Flujo de datos:



Tiempo:

	Ciclos M	*Estados T*	*µseg @ 2 MHz*
condición cierta	5	17	8.5
condición falsa	3	10	5

Direccionamiento: Inmediato

Códigos byte:

CC	NZ	Z	NC	C	PO	PE	P	M	
	C4	CC	D4	DC	E4	EC	F4	FC	-q-p

Banderas:

S	Z		H		P/V	N	C

(efecto nulo)

Ejemplo: CALL Z, B042

Antes:

F: 85

PC: 0B01

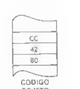
SP: BB12

Después:

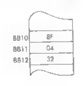
F: 85

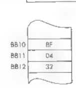
PC: 0B04

SP: BB12



CODIGO OBJETO





CALL pq

Función:

Formato:

Descripción:

Flujo de datos:

Tiempo:

Direccionamiento:

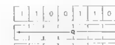
Banderas:

CALL pq

Llamada a subrutina a la posición pq.

Función: $(SP - 1) \leftarrow PC_{sup}$; $(SP - 2) \leftarrow PC_{inf}$; $SP \leftarrow SP - 2$; $PC \leftarrow pq$

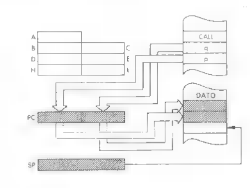
Formato:



byte 1: CD
byte 2: dirección, byte inferior
byte 3: dirección, byte superior

Descripción: El contenido del contador del programa se empuja en la pila tal como se describe en las instrucciones PUSH. A continuación se carga el contenido de la posición de memoria siguiente al código de operación en la mitad inferior del PC y el de la posición siguiente en la mitad superior del PC. La instrucción siguiente se traerá de esta nueva dirección.

Flujo de datos:



Tiempo: 5 ciclos M; 17 estados T; 8.5 μseg @ 2 MHz.

Direccionamiento: Inmediato.

Banderas:



(efecto nulo)

Ejemplo: CALL 40B1

Antes:

PC AA40

SP 0B14

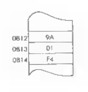
Después:

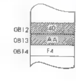
PC 40B1

SP 0B12

CD
B1
40

CODIGO OBJETO





CCF

Función:

Formato:

Descripción:

Flujo de datos:

Tiempo:

Direccionamiento:

Banderas:

CCF

Complementación de la bandera de acarreo.

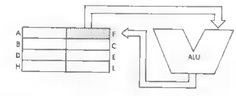
Función: $C \leftarrow \overline{C}$

Formato:

0	0	1	1	1	1	1	1	3F

Descripción: Se complementa la bandera de acarreo.

Flujo de datos:



Tiempo: 1 ciclo M; 4 estados T; 2 μseg @ 2 MHz.

Direccionamiento: Implícito.

Banderas:

S	Z		H		P/V	N	C
			?			0	●

CP s

Comparación del operando s con el acumulador.

Función: A – s

Formato: s puede ser r, n, (HL), (IX + d) o (IY + d).

r	1 0 1 1 1 ←r→	
n	1 1 1 1 1 1 1 0	FE
	←n→	byte 2: dato inmediato
(HL)	1 0 1 1 1 1 1 0	byte 1: BE
(IX + d)	1 1 0 1 1 1 0 1	byte 1: DD
	1 0 1 1 1 1 1 0	byte 2: BE
	←d→	byte 3: valor del desplazamiento
(IY + d)	1 1 1 1 1 1 0 1	byte 1: FD
	1 0 1 1 1 1 1 0	byte 2: BE
	←d→	byte 3: valor del desplazamiento

r puede ser:

A – 111	E – 011
[redacted] – 000	H – 100
C – 001	L – 101
D – 010	

Descripción: El operando especificado se resta del acumulador, y el resultado se descarta; s se define en la descripción de instrucciones ADD y similares.

Flujo de datos:

Tiempo:

Direccionamiento:

Códigos byte:

Banderas:

Ejemplo:



■ el acumulador.

o (IY + d).

dato inmediato

BE

DD

BE

valor del desplazamiento

FD

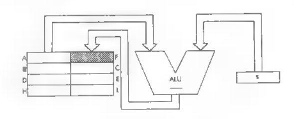
BE

valor del desplazamiento

E – 011
H – 100
L – 101

el acumulador, y el resultado
pción de instrucciones ADD

Flujo de datos:



Tiempo:

s	Ciclos M	Estados T	μseg @ 2 MHz
r	1	4	2
■	2	7	3.5
(HL)	2	7	3.5
(IX + d)	5	19	9.5
(IY + d)	5	19	9.5

Direccionamiento: r: implícito; ■: inmediato; (HL): indirecto; (IX + d), (IY + d): indexado.

Códigos byte: CP r:

r	A	B	C	D	E	H	L
	BF	■	■	BA	BB	BC	BD

Banderas:

S	Z		H		P/V	N	C
●	●		●		●	1	●

Ejemplo: CP (HL)

Antes:

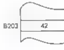
A 96 | 36 F

H B203 L

Después:







CPD

Comparación con decremento.

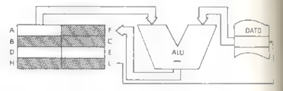
Función: A − [HL]; HL ← HL − 1; BC ← BC − 1

Formato:

1	1	1	0	1	1	0	1	byte 1: ED
1	0	1	0	1	0	0	1	byte 2: A9

Descripción: El contenido de la posición de memoria direccionada por el par de registros HL se resta del contenido del acumulador y se descarta el resultado. A continuación se decrementan los pares de registros HL y BC.

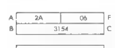
Flujo de datos:



Tiempo: 4 ciclos M; 16 estados T; [redacted] µseg @ 2 MHz.

Direccionamiento: Indirecto.

Banderas:



Poner a 0 [redacted] BC = 0 tras la ejecución; dejar a 1 en caso contrario.
Hacer 1 si A = (HL)

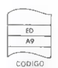
Ejemplo: CPD

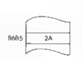
Antes:

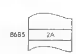
A	2A	06	F
B	3154		C
H	86B5		L

Después:

A	2A	46	F
B	3153		C
H	86B4		L







CPDR

Función:

Formato:

Descripción:

Flujo de datos:

Tiempo:

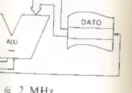
Banderas:

– BC – 1

ED

A9

moria direccionada por el par
tenido del acumulador y se
ión se decrementan los pares



@ 2 MHz.

BC = 0 tras la ejecución;
a caso contrario.
= (HL)

pués:

CPDR

Comparación de bloques con decremento.

Función:

A – [HL]; HL ← HL – 1; BC ← BC – 1;
Repetir hasta que BC = 0 o A = (HL).

Formato:

1	1	1	0	1	1	0	1	byte 1: ED
1	0	1	1	1	0	0	1	byte 2: B9

Descripción:

El contenido de la posición de memoria direccionada por el par de registros HL se resta del contenido del acumulador y el resultado se descarta. A continuación se decrementan los pares de registros BC y HL. Si BC ≠ 0 y A ≠ (HL), el contador del programa se decrementa en dos, y la instrucción vuelve a ejecutarse.

Flujo de datos:

A B D H F C E L ALU DATO

Tiempo:

BC = 0 o A = (HL): 4 ciclos M; 16 estados T; 8 μseg @ 2 MHz.
BC ≠ 0 y A ≠ (HL): 5 ciclos M; 21 estados T; 10.5 μseg @ 2MHz.

Banderas:

S Z H P/V N C

Poner a 0 si BC = 0 tras la ejecución; deja a 1 en caso contrario.

Dejar a 1 si A = (HL).

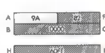


Ejemplo: CPDR

Antes:

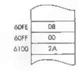
A	9A	00	F
B	0002		C
H	6100		L

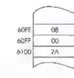
Después:



ED	
B9	

CODIGO OBJETO





CPI

Función:

Formato:

Descripción:

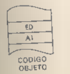
Flujo de datos:

Tiempo:

Direccionamiento:

Banderas:

Ejemplo:



CPI

Comparación con incremento.

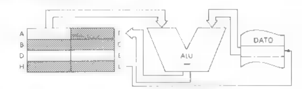
Función: A – [HL]; HL ← HL + 1; BC ← BC – 1

Formato:

1	1	1	0	1	1	0	1	byte 1: ED
1	0	1	0	0	0	0	1	byte 2: A1

Descripción: El contenido de la posición de memoria direccionada por el par de registros HL se resta del contenido del acumulador y se descarta el resultado. El par de registros HL se incrementa, y el BC se decrementa.

Flujo de datos:



Tiempo: 4 ciclos M; 16 estados T; 8 μseg @ 2 MHz.

Direccionamiento: Indirecto.

Banderas:

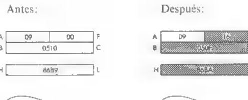
S	Z		H		P/V	N	C
●	X		●		X	1	

Poner a 0 si BC = 0 tras la ejecución; dejar a 1 en caso contrario.

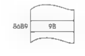
Dejar a 1 si A = (HL).

Ejemplo: CPI

Antes: Después:









CPIR

Comparación de bloques con incremento.

Función:

A − [HL]; HL ← HL + 1; BC ← BC − 1;
Repetir hasta que BC = 0 o A = (HL).

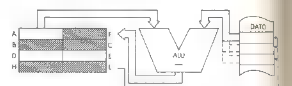
Formato:

1	1	1	0	1	1	0	1	byte 1: ED
1	0	1	1	0	0	0	1	byte 2: B1

Descripción:

El contenido de la posición de memoria direccionada por el p de registros HL se resta del contenido del acumulador y resultado se descarta. A continuación se incrementa el par registros HL y se decrementa el BC. Si BC ≠ 0 y A ≠ (HL), contador del programa se decrementa en dos y la instrucc vuelve a ejecutarse.

Flujo de datos:



Tiempo:

BC = 0 o A = (HL): 4 ciclos M; 16 estados T: 8 μseg @ 2 MHz.
BC ≠ 0 y A ≠ (HL): 5 ciclos M; 21 estados T: 10.5 μseg @ 2 MHz.

Direccionamiento:

Indirecto.

Banderas:

S	Z		H		P/V	N	C
●	X		●		X	1	

Poner a 0 si BC = 0 tras la ejecución; fijar a 1 en caso contrario

Fijar a 1 si A = (HL)

Ejemplo:

ED
B1
CODIGO OBJETO

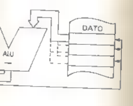
ıcremento.

← BC − 1;
= (HL).

ED

B1

moria direccionada por el par
ntenido del acumulador y el
ción se incrementa el par de
C. Si BC ≠ 0 y A ≠ (HL), el
enta en dos y la instrucción



16 estados T: 8 μseg @ 2

1 estados T: 10.5 μseg @ 2

0 si BC = 0 tras la eje-
ijar a 1 en caso contrario.
si A = (HL)

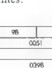


Ejemplo:

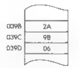
CPIR

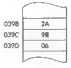
Antes:

Después:




ED
B1
CODIGO OBJETO





CPL

Complementar el acumulador.

Función: $A \leftarrow \overline{A}$

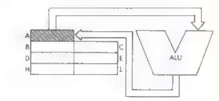
Formato:

0	0	1	0	1	1	1	1

2F

Descripción: El contenido del acumulador se complementa (se invierte) y resultado vuelve a almacenarse en el acumulador (compleme to a 1).

Flujo de datos:



Tiempo: 1 ciclo M; 4 estados T; 2 μseg @ 2 MHz.

Direccionamiento: Implícito.

Banderas:

S	Z		H		P/V	N	C
			1			1	

Ejemplo: CPL

Antes: A 3D

Después: A C2



DAA

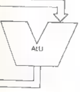
Función:

Formato:

Descripción:

Flujo de datos:

omplementa (se invierte) y el
el acumulador (complemen-



@ 2 MHz.

spués:

DAA

Ajuste decimal del acumulador.

Función: Véase a continuación.

Formato:

0	0	1	0	0	1	1	1

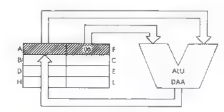
27

Descripción:

La instrucción suma condicionalmente "6" al *nibble* derecho, o al izquierdo, o a los dos, del acumulador, según el registro de estado, para realizar la conversión BCD tras las operaciones aritméticas.

N	*C*	*Valor del* nibble *sup.*	*H*	*Valor del* nibble *inf.*	*Número sumado a A*	*C tras la ejecución*
0 (ADD, ADC, INC)	0	0-9	0	0-9	00	0
	0	0-8	0	A-F	06	0
	0	0-9	1	0-3	06	0
	0	A-F	0	0-9	60	1
	0	9-F	0	A-F	66	1
	0	A-F	1	0-3	66	1
	1	0-2	0	0-9	60	1
	1	0-2	0	A-F	66	1
	1	0-3	1	0-3	66	1
1 (SUB, SBC, DEC, NEG)	0	0-9	0	0-9	00	0
	0	0-8	1	6-F	FA	0
	1	7-F	0	0-9	AO	1
	1	6-F	1	6-F	9A	1

Flujo de datos:



Tiempo: 1 ciclo M; 4 estados T; 2 μseg @ 2 MHz.

Direccionamiento: Implícito.

Banderas:

S	Z		H		P/V	N	C
●	●		●		●		●

Ejemplo: DAA

CODIGO OBJETO: 27

	Antes:		Después:	
	A	F	A	F
	■	94	16	05

DEC m

Función:

Formato:

r

(HL)

(IX + d)

(IY + d

Descripción:

Flujo de datos:

DEC m

Decrementa el operando m.

Función: m ← m − 1

Formato: m puede ser r, (HL), (IX + d), (IY + d).

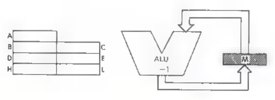
r	0	0	←	r	→	1	0	1	
(HL)	0	0	1	1	0	1	0	1	35
(IX + d)	1	1	0	1	1	1	0	1	byte 1: DD
	0	0	1	1	0	1	0	1	byte 2: 35
	←			d				→	byte 3: valor del desplazamiento
(IY + d)	1	1	1	1	1	1	0	1	byte 1: FD
	0	0	1	1	0	1	0	1	byte 2: 35
	←			d				→	byte 3: valor del desplazamiento

r puede ser:

A − 111	E − 011
B − 000	H − 100
C − 001	L − 101
D − 010	

Descripción: El contenido de la posición direccionada por el operando especificado se decrementa y vuelve a almacenarse en esa posición: m se define en la descripción de instrucciones INC similares.

Flujo de datos:



Tiempo:

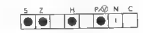
m	*Ciclos M*	*Estados T*	*μseg @ 2 MHz*
r	1	4	2
(HL)	3	11	5.5
(IX + d)	6	23	11.5
(IY + d)	6	23	11.5

Direccionamiento: r: implícito; (HL): indirecto; (IX + d), (IY + d): indexado.

Códigos byte: DEC r

r:	A	B	C	D	E	H	L
	3D	05	0D	15	1D	25	2D

Banderas:



Ejemplo: DEC C

Antes:



Después:

0E C



DEC rr

Función:

Formato:

Descripción:

Flujo de datos:

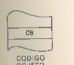
Tiempo:

Direccionamiento:

Códigos byte: rr

Banderas:

Ejemplo:



DEC rr

Decrementa el par de registros rr.

Función: rr ← rr − 1

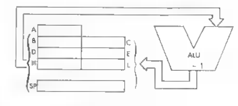
Formato:

0	0	r	r	1	0	1	1

Descripción: Se decrementa el contenido del par de registros especificado, y el resultado vuelve a almacenarse en ese mismo par; rr puede ser:

BC – 00	HL – 10
DE – 01	SP – 11

Flujo de datos:



Tiempo: 1 ciclo M; 6 estados T; 3 μseg @ 2 MHz.

Direccionamiento: Implícito.

Códigos byte:

rr:

BC	DE	HL	SP
0B	1B	2B	3B

Banderas:



(efecto nulo)

Ejemplo:

DEC BC

Antes: B 3B11 C

Después:





DEC IX

Decrementa IX.

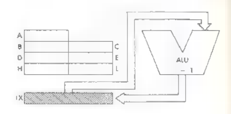
Función: IX ← IX − 1

Formato:

1	1	0	1	1	1	0	1	byte 1: DD
0	0	1	0	1	0	1	1	byte 2: 2B

Descripción: Se decrementa el contenido del registro IX, y el resultado vuelve a almacenarse en IX.

Flujo de datos:



Tiempo: 2 ciclos M; 10 estados T; 5 μseg @ 2 MHz.

Direccionamiento: Implícito.

Banderas:

S	Z		H		P/V	N	C

(efecto nulo)

Ejemplo: DEC IX

Antes: IX 6114

Después: IX 6113

DD
2B

CODIGO OBJETO

DEC IY

Función:

Formato:

Descripción:

Flujo de datos:

Tiempo:

Direccionamiento:

Banderas:

Ejemplo:



CODIGO OBJETO

DEC IY

Decrementa IY.

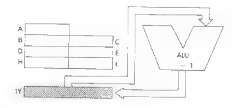
Función: IY ← IY − 1

Formato:

1	1	1	1	1	1	0	1	byte 1: FD
0	0	1	0	1	0	1	1	byte 2: 2B

Descripción: Se decrementa el contenido del registro IY, y el resultado vuelve a almacenarse en IY.

Flujo de datos:

Tiempo: 2 ciclos M; 10 estados T; 5 μseg @ 2 MHz.

Direccionamiento: Implícito.

Banderas:

S	Z		H		P/V	N	C

(efecto nulo)

Ejemplo: DEC IY

Antes: IY 900F

Después: IY 900E

FD
2B
CODIGO OBJETO

DI

Invalida interrupciones.

Función: IFF ← 0

Formato:

1	1	1	1	0	0	1	1

F3

Descripción: El biestable de interrupciones se pone a 0 y, en consecuencia, impiden todas las interrupciones enmascarables. Vuelve a ha- darse con una instrucción EI.

Tiempo: 1 ciclo M; 4 estados T; 2 μseg @ 2 MHz.

Direccionamiento: Implícito.

Banderas:

S	Z		H		P/V	N	C

(efecto nulo)

DJNZ ■

Función:

Formato:

Descripción:

Flujo de datos:

Tiempo:

Direccionamiento:

Banderas:

Ejemplo:



one a 0 y, en consecuencia, se
enmascarables. Vuelve ■ vali-

@ 2 MHz.

ulo)

DJNZ e

Decremento de B y, si no es cero, salto relativo de longitud e.

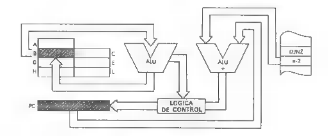
Función: B ← B − 1; si B ≠ 0: PC ← PC + e

Formato:

0	0	0	1	0	0	0	0	byte 1: 10
e-2								byte 2: valor del desplazamiento

Descripción: Se decrementa el registro B. Si el resultado no es cero, se suma el valor del desplazamiento al contador del programa en aritmética de complemento ■ dos, para permitir saltos hacia adelante y hacia atrás. El valor del desplazamiento se suma al PC + 2 (tras el salto); en consecuencia, el desplazamiento efectivo va de −126 a +129 bytes. El ensamblador resta automáticamente del valor del desplazamiento fuente para generar el código hexadecimal.

Flujo de datos:



Tiempo: B ≠ 0: 3 ciclos M; 13 estados T; 6.5 μseg @ 2 MHz.
B = 0: 2 ciclos M; 8 estados T; 4 μseg @ 2 MHz.

Direccionamiento: Inmediato.

Banderas:



(efecto nulo)

Ejemplo: DJNZ $ − 5 ($ = PC actual)

Antes: Después:



51 B

PC 0DE1



EI

Habilita interrupciones.

Función: IFF ← 1

Formato:

1	1	1	1	1	0	1	1

FB

Descripción: El biestable de interrupciones se pone a 1 y, en consecuencia permite que se produzcan interrupciones enmascarables tras la ejecución de la instrucción que sigue a la EI. En ese tiempo se invalidan las interrupciones filtrables.

Tiempo: 1 ciclo M; 4 estados T; 2 μseg @ 2 MHz.

Direccionamiento: Implícito.

Banderas:

S	Z		H		P/V	N	C

(efecto nulo)

Ejemplo: Una secuencia habitual al término de una rutina de interrupción sería:

EI
RETI

La interrupción enmascarable vuelve a habilitarse al término de RETI.

EX AF, AF'

Función:

Formato:

Descripción:

Flujo de datos:

Tiempo:

Direccionamiento:

Banderas:

Ejemplo:



pone a 1 y, en consecuencia,
ciones enmascarables tras la
sigue a la EI. En ese tiem-
filtrables.

@ 2 MHz.

ulo)

de una rutina de interrup-

e a habilitarse al término de

EX AF, AF'

Intercambia acumulador y banderas con los registros alternativos.

Función: AF ↔ AF'

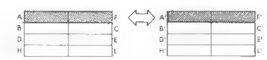
Formato:

0	0	0	0	1	0	0	0

08

Descripción: Los contenidos del acumulador y del registro de estado se intercambian con los del acumulador y el registro de estado alternativos.

Flujo de datos:



Tiempo: 1 ciclo M; 4 estados T; 2 μseg @ 2 MHz.

Direccionamiento: Implícito

Banderas:

S	Z		H		P/V	N	C
●	●	●	●	●	●	●	●

Ejemplo: EX AF, AF'



Antes: Después:



EX DE, HL

Intercambia los registros HL y DE.

Función: DE ⟷ HL

Formato:

1	1	1	0	1	0	1	1

EB

Descripción: Se intercambian los contenidos de los dos pares de registros DE y HL.

Flujo de datos:

A, B C, D E, H L

Tiempo: 1 ciclo M; 4 estados T; 2 μseg @ 2 MHz.

Direccionamiento: Implícito.

Banderas: S Z H P/V N C (efecto nulo)

Ejemplo: EX DE, HL

Antes:

D	A4E6	E
H	9604	L

Después:

D	9604	E
H	A4E6	L

EB

CODIGO OBJETO

EX (SP), HL

Función:

Formato:

Descripción:

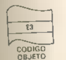
Flujo de datos:

Tiempo:

Direccionamiento:

Banderas:

Ejemplo:



EX (SP), HL

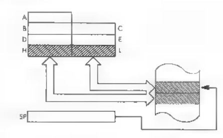
Intercambia HL con el elemento superior de la pila.

Función: (SP) ↔ L; (SP + 1) ↔ H

Formato: | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | E3

Descripción: El contenido del registro L se intercambia con el de la posición de memoria direccionada por el puntero de la pila. El contenido del registro H se intercambia con el de la posición de memoria inmediatamente siguiente a la direccionada por el puntero de la pila.

Flujo de datos:



Tiempo: 5 ciclos M; 19 estados T; 9.5 μseg @ 2 MHz.

Direccionamiento: Indirecto.

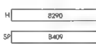
Banderas:

S	Z		H		P/V	N	C

(efecto nulo)

Ejemplo: EX (SP), HL

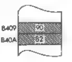
Antes:





Después:







EX (SP), IX

Intercambia IX con el elemento superior de la pila.

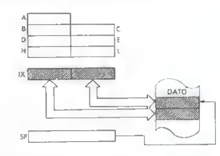
Función: $(SP) \leftrightarrow IX_{inf}$; $(SP + 1) \leftrightarrow IX_{sup}$

Formato:

1	1	0	1	1	1	0	1	byte 1: DD
1	1	1	0	0	0	1	1	byte 2: E3

Descripción: El contenido de la mitad inferior del registro IX se intercambia con el de la posición de memoria direccionada por el apuntador de la pila. El de la mitad superior de IX se intercambia, a su vez, con el de la posición de memoria inmediatamente siguiente a la direccionada por el apuntador de la pila.

Flujo de datos:



Tiempo: 6 ciclos M; 23 estados T; 11.5 μseg @ 2 MHz.

Direccionamiento: Indirecto.

Banderas:



(efecto nulo)

Ejemplo:



EX

An

IX

SP

0402

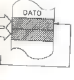
0403

superior de la pila.

DD

E3

del registro IX se intercambia
ireccionada por el apuntador
r de IX se intercambia, ■ su
ria inmediatamente siguiente
or de la pila.



g @ 2 MHz.

lo)

Ejemplo: EX (SP), IX

Antes:

IX 9234

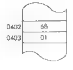
SP 0402

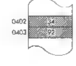
Después:

IX 0168

SP 0402







EX (SP), IY

Intercambia IY con la parte superior de la pila.

Función: $(SP) \leftrightarrow IY_{inf}$; $(SP + 1) \leftrightarrow IY_{sup}$

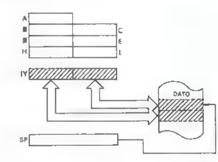
Formato:

1	1	1	1	1	1	0	1	byte 1: FD

1	1	1	0	0	0	1	1	byte 2: E3

Descripción: El contenido de la mitad inferior del registro IY se intercambia con el de la posición de memoria direccionada por el apuntador de la pila. El de la mitad superior de IY se intercambia, a su vez, con el de la posición de memoria inmediatamente siguiente a la direccionada por el apuntador de la pila.

Flujo de datos:



Tiempo: [illegible] ciclos M; 23 estados T; 11.5 μseg @ 2 MHz.

Direccionamiento: Indirecto.

Banderas:



(efecto nulo)

Ejemplo:

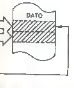


rior de la pila.

FD

E3

el registro IY se intercambia
reccionada por el apuntador
de IY se intercambia, a su
ria inmediatamente siguiente
r de la pila.

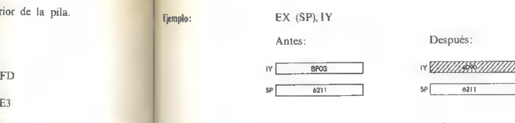


g @ 2 MHz.

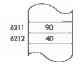
lo)

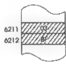
Ejemplo: EX (SP), IY

Antes: Después:









EXX

Intercambia registros alternativos.

Función: BC ↔ BC'; DE ↔ DE'; HL ↔ HL'

Formato: | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | D9

Descripción: Los contenidos de los registros de tipo general se intercambian con los correspondientes registros alternativos.

Flujo de datos:

A F / B C / D E / H L ⇄ A' F' / B' C' / D' E' / H' L'

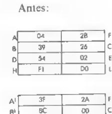
Tiempo: 1 ciclo M; 4 estados T; 2 μseg @ 2 MHz.

Direccionamiento: Implícito.

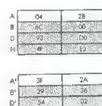
Banderas: S Z H P/V N C (efecto nulo)

Ejemplo: EXX

Antes:



Después:



D9

CODIGO OBJETO

HALT

De

Función: La

Formato:

Descripción:

Tiempo:

Direccionamiento:

Banderas:

HALT

Detiene la CPU.

Función: La CPU deja de actuar.

Formato:

0	1	1	1	0	1	1	0

76

Descripción: La CPU deja de funcionar y ejecuta instrucciones NOP continuamente, para proseguir con los ciclos de refresco de la memoria, hasta que recibe una interrupción o una orden de reinicio.

Tiempo: 1 ciclo M; 4 estados T; 2 μseg @ 2 MHz + el tiempo necesario para ejecutar un número indefinido de instrucciones NOP.

Direccionamiento: Implícito.

Banderas:

S	Z		H		P/V	N	C

(efecto nulo)

IM 0

Activa el modo de interrupción 0.

Función: Control interno de interrupciones.

Formato:

1	1	1	0	1	1	0	1	byte 1: ED
0	1	0	0	0	1	1	0	byte 2: 46

Descripción: Activa el modo de interrupción 0. En esta situación, el dispositivo interruptor puede dejar una instrucción en el *bus* de datos para su ejecución; el primer byte de dicha instrucción debe aparecer durante el ciclo de identificación de la interrupción.

Tiempo: 2 ciclos M; 8 estados T; 4 μseg @ 2 MHz.

Direccionamiento: Implícito.

Banderas:

S	Z		H		P/V	N	C

(efecto nulo)

IM 1

Función:

Formato:

Descripción:

Flujo de datos:

Tiempo:

Direccionamiento:

Banderas:

IM 1

Activa el modo de interrupción 1.

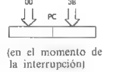
Función: Control interno de interrupciones.

Formato:

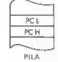
1	1	1	0	1	1	0	1	byte 1: ED
0	1	0	1	0	1	1	0	byte 2: 56

Descripción: Activa el modo de interrupción 1. Cuando se produce la interrupción, se ejecuta una instrucción RST 0038H.

Flujo de datos:



(en el momento de la interrupción)



PCL
PCH
PILA

Tiempo: 2 ciclos M; 8 estados T; 4 μseg @ 2 MHz.

Direccionamiento: Implícito.

Banderas:

S	Z		H		P/V	N	C

(efecto nulo)

IM 2

Activa el modo de interrupción 2.

Función: Control interno de interrupciones.

Formato:

1	1	1	0	1	1	0	1	byte 1: ED

0	1	0	1	1	1	1	0	byte 2: 5E

Descripción: Activa el modo de interrupción 2. Cuando se produce la interrupción, el periférico utilizado debe entregar un byte como parte inferior de una dirección; la parte superior del vector de dirección procede del contenido del registro I. Este señala una segunda dirección almacenada en memoria, que se carga en el contador del programa, tras lo cual comienza la ejecución.

Tiempo: 2 ciclos M; 8 estados T; 4 μseg @ 2 MHz.

Direccionamiento: Implícito.

Banderas:

S	Z		H		P/V	N	C

(efecto nulo)

IN r,(C)

Carga el registro r a partir del puerto (C).

Función: r ← (C)

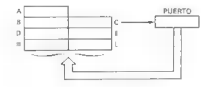
Formato:

1	1	1	0	1	1	0	1	byte 1: ED
0	1	←	r	→	0	0	0	byte 2

Descripción:

Se lee el dispositivo periférico direccionado por el contenido del registro C, y el resultado se carga en el registro especificado. C proporciona los bits A0 a A7 del *bus* de direcciones. B proporciona los bits A8 a A15.

Flujo de datos:



r puede ser:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

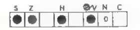
Tiempo: 3 ciclos M; 12 estados T; [illegible] μseg @ 2 MHz.

Direccionamiento: Externo.

Códigos byte:

r:	A	B	C	D	E	H	L
ED	78	40	48	50	58	60	68

Banderas:



Es importante señalar que IN A,(N) no ejerce ningún efecto sobre las banderas, al contrario que IN r,(C), que sí lo ejerce.

Ejemplo: IN D,(C)

Antes:

C: A5
D: 09
PUERTO: 6A
A6

ED
50
CODIGO OBJETO

Después:

C: A5
D: 6A
PUERTO: 6A
A5

IN A, (N)

Función:

Formato:

Descripción:

Flujo de datos:

Tiempo:

Direccionamiento:

Banderas:

Ejemplo:

DB
B2
CODIGO OBJETO

IN A,(N)

Carga el acumulador a partir del puerto de entrada N.

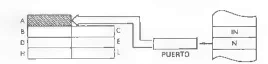
Función: A ← (N)

Formato:

1	1	0	1	1	0	1	1	byte 1: DB
←			N				→	byte 2: dirección puerto

Descripción: Se lee el dispositivo periférico N, y el resultado se carga en el acumulador.
El literal N se sitúa en las líneas A0 a A7 del *bus* de direcciones. A proporciona los bits A8 a A15.

Flujo de datos:



Tiempo: 3 ciclos M; 11 estados T; 5.5 μseg @ 2 MHz.

Direccionamiento: Externo.

Banderas: S Z H P/V N C (efecto nulo)

Ejemplo: IN A,(B2)

Antes: A B4, PUERTO F1 (B2)

Después: A F1, PUERTO F1 (B2)

CODIGO OBJETO: DB, B2

INC r

Incrementa el registro r.

Función: r ← r + 1

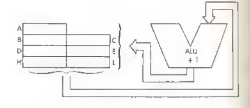
Formato:

0	0	←	r	→	1	0	0

Descripción: El contenido del registro especificado se incrementa; r puede ser:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Flujos de datos:



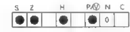
Tiempo: 1 ciclo M; 4 estados T; 2 μseg @ 2 MHz.

Direccionamiento: Implícito.

Códigos byte:

r:	A	B	C	D	E	H	L
	3C	04	0C	14	1C	24	2C

Banderas:



Ejemplo: INC D

Antes: D 06

Después:





INC rr

Función:

Formato:

Descripción:

Flujo de datos:

Tiempo:

Direccionamiento:

Códigos byte:

Banderas:

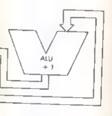
Ejemplo:

23

CODIGO
OBJETO

cado se incrementa; r puede

E – 011
H – 100
L – 101



2 MHz.

ués:

INC rr

Incrementa el par de registros rr.

Función: rr ← rr + 1

Formato:

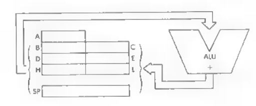
0	0	r	r	0	0	1	1

Descripción: El contenido del par de registros especificado se incrementa, y el resultado se almacena de nuevo en el mismo par; rr puede ser:

BC – 00 HL – 10
DE – 01 SP – 11

Flujo de datos:





Tiempo: 1 ciclo M; 6 estados T; 3 μseg @ 2 MHz.

Direccionamiento: Implícito.

Códigos byte:

rr:

BC	DE	HL	SP
03	13	23	33

Banderas:



(efecto nulo)

Ejemplo: INC HL

Antes: Después:







INC (HL)

Incrementa la posición de memoria direccionada indirectam (HL).

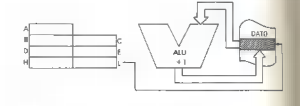
Función: (HL) ← (HL) + 1

Formato:

0	0	1	1	0	1	0	0	34

Descripción: Se incrementa el contenido de la posición de memoria direc nada por el par de registros HL, y el resultado se almacena nuevo en dicha posición.

Flujo de datos:



Tiempo: 3 ciclos M; 11 estados T; 5.5 μseg @ 2 MHz.

Direccionamiento: Indirecto.

Banderas:

S	Z		H		P/V	N	C
●	●		●		●	0	

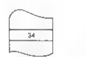
Ejemplo: INC (HL)

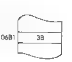
Antes:

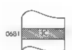
H 06B1 L

Después:

H 06B1 L







INC (IX + d)

In

Función:

Formato:

Descripción:

Flujo de datos:

Tiempo:

Direccionamiento:

Banderas:

ria direccionada indirectamente

posición de memoria direccio-
y el resultado se almacena de

eg @ 2 MHz.

pués:

0681



INC (IX + d)

Incrementa la posición de memoria indexada (IX + d).

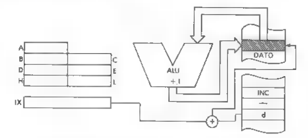
Función: (IX + d) ← (IX + d) + 1

Formato:

1	1	0	1	1	1	0	1	byte 1: DD
0	0	1	1	0	1	0	0	byte 2: 34
d								byte 3: valor del desplazamiento

Descripción: Se incrementa el contenido de la posición de memoria direccionada por el registro IX más el valor del desplazamiento dado, y el resultado vuelve a almacenarse en dicha posición.

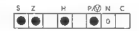
Flujo de datos:



Tiempo: 6 ciclos M; 23 estados T; 11,5 μseg @ MHz.

Direccionamiento: Indexado.

Banderas:



Ejemplo: INC (IX + 2)

Antes: Después:

IX 03B1 IX 03B1

CODIGO OBJETO
DD
34
02

	Antes	Después
03B1	B1	B1
03B2	85	85
03B3	B9	BA

INC (IY + d)

Función:

Formato:

Descripción:

Flujo de datos:

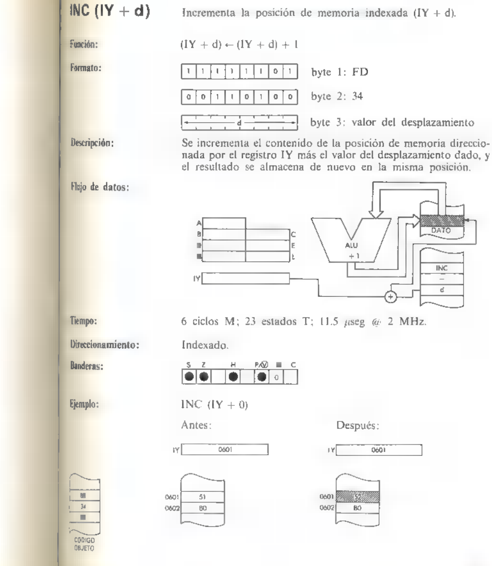
Tiempo:

Direccionamiento:

Banderas:

Ejemplo:

CODIGO OBJETO
FD
34
00



INC (IY + d)

Incrementa la posición de memoria indexada (IY + d).

Función: (IY + d) ← (IY + d) + 1

Formato:

1	1	1	1	1	1	0	1	byte 1: FD
0	0	1	1	0	1	0	0	byte 2: 34
←			d				→	byte 3: valor del desplazamiento

Descripción: Se incrementa el contenido de la posición de memoria direccionada por el registro IY más el valor del desplazamiento dado, y el resultado se almacena de nuevo en la misma posición.

Flujo de datos:

Tiempo: 6 ciclos M; 23 estados T; 11.5 μseg @ 2 MHz.

Direccionamiento: Indexado.

Banderas:

S	Z		H		P/V	■	C
●	●		●		●	0	

Ejemplo: INC (IY + 0)

Antes:

Después:

INC IX

Incrementa IX.

Función: IX ← IX + 1

Formato:

1	1	0	1	1	1	0	1

byte 1: DD

0	0	1	0	0	0	1	1

byte 2: 23

Descripción: Se incrementa el contenido del registro IX, y el resultado vuelve a almacenarse en IX.

Flujo de datos:

A B D H C E L ALU + 1 IX

Tiempo: 2 ciclos M; 10 estados T; 5 μseg @ 2 MHz.

Direccionamiento: Implícito.

Banderas:

S Z H P/V N C

(efecto nulo)

Ejemplo:



INC IX

Antes: Después:



INC IY

Función:

Formato:

Descripción:

Flujo de datos:

Tiempo:

Direccionamiento:

Banderas:

Ejemplo:

FD
23
CODIGO OBJETO

INC IY

Incrementa IY.

Función:

IY ← IY + 1

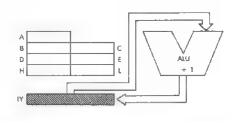
Formato:

1	1	1	1	1	1	0	1	byte 1: FD
0	0	1	0	0	0	1	1	byte 2: 23

Descripción:

Se incrementa el contenido del registro IY, y el resultado vuelve a almacenarse en IY.

Flujo de datos:



Tiempo:

2 ciclos M; 10 estados T; 5 μseg @ 2 MHz.

Direccionamiento:

Implícito.

Banderas:



(efecto nulo)

Ejemplo:

INC IY

Antes:



Después:





IND

Entrada con decremento.

Función: (HL) ← (C); B ← B − 1; HL ← HL − 1

Formato:

1	1	1	0	1	1	0	1	byte 1: ED
1	0	1	0	1	0	1	0	byte 2: AA

Descripción: Se lee el dispositivo periférico direccionado por el registro C, el resultado se carga en la posición de memoria direccionada por el par de registros HL. A continuación se decrementan el registro B y el par HL.

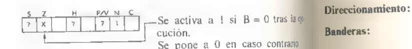
Flujo de datos:



Tiempo: 4 ciclos M; 16 estados T; 8 μseg @ 2 MHz.

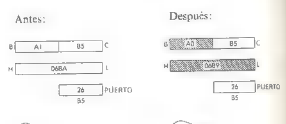
Direccionamiento: Externo.

Banderas:

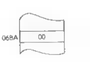


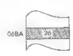
Se activa a 1 si B = 0 tras la ejecución.
Se pone a 0 en caso contrario

Ejemplo: IND









INDR

Función:

Formato:

Descripción:

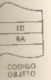
Flujo de datos:

Tiempo:

Direccionamiento:

Banderas:

Ejemplo:



INDR

Entrada de bloque con decremento.

Función:

(HL) ← (C); B ← B − 1; HL ← HL − 1
Repetir hasta que B = 0

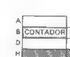
Formato:

1	1	1	0	1	1	0	1	byte 1: ED
1	0	1	1	1	0	1	0	byte 2: BA

Descripción:

Se lee el dispositivo periférico direccionado por el registro C, y el resultado se carga en la posición de memoria direccionada por el par de registros HL. A continuación se decrementan el registro B y el par HL. Si B no es cero, el contador del programa se decrementa en 2, y la instrucción vuelve a ejecutarse.

Flujo de datos:



Tiempo:

B = 0: 4 ciclos M; 16 estados T; 8 μseg @ 2 MHz.
B ≠ 0: 5 ciclos M; 21 estados T; 10.5 μseg @ 2 MHz.

Direccionamiento: Externo.

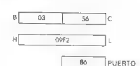
Banderas:

S	Z		H		P/V	N	C
?	1		?		?	1	

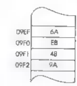
Ejemplo:

INDR

Antes:



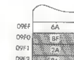




Después:









INI

Entrada con incremento.

Función: (HL) ← (C); B ← B − 1; HL ← HL + 1

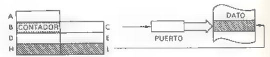
Formato:

1	1	1	0	1	1	0	1	byte 1: ED
1	0	0	1	0	0	1	0	byte 2: A2

Descripción: Se lee el dispositivo periférico direccionado por el registro C, y el resultado se carga en la posición de memoria direccionada por el par de registros HL. El registro B se decrementa y el par de registros HL se incrementa.

El contenido de C se deja en la mitad inferior del bus de direcciones y el de B en la superior. La selección de E/S suele hacerse mediante C, es decir, mediante A0 a A7. B es el contador de byte.

Flujo de datos:



Tiempo: 4 ciclos M; 16 estados T; [redacted] μseg @ 2 MHz.

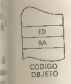
Direccionamiento: Externo.

Banderas:

S	Z		[redacted]		P/V	N	C
?	X		?		?	1	

Z se activa a 1 si B = 0 tras la ejecución. Se pone a 0 en caso contrario.

Ejemplo:



IL + 1

ED

A2

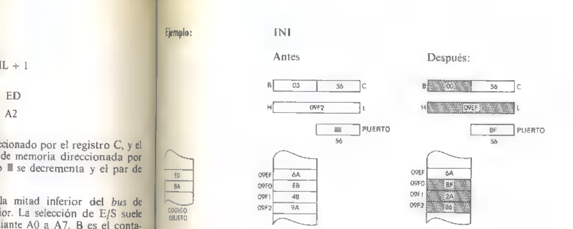
cionado por el registro C, y el
de memoria direccionada por
se decrementa y el par de

la mitad inferior del *bus* de
ior. La selección de E/S suele
iante A0 a A7. B es el conta-

g @ 2 MHz.

ejecución.

Ejemplo: INI



INIR

Entrada de bloque con incremento.

Función:

(HL) ← (C); B ← B − 1; HL ← HL + 1; repetir hasta que B = 0

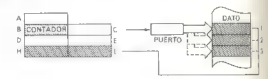
Formato:

1	1	1	0	1	1	0	1	byte 1: ED
1	0	1	1	0	0	1	0	byte 2: B2

Descripción:

Se lee el dispositivo periférico direccionado por C, y el resultado se carga en la posición de memoria direccionada por el par de registros HL. El registro B se decrementa y el par HL se incrementa. Si B no es cero, el contador del programa se decrementa en 2, y la instrucción vuelve a ejecutarse.

Flujo de datos:



Tiempo:

B = 0: 4 ciclos M: 16 estados T: 8 μseg @ 2 MHz.
B ≠ 0: 5 ciclos M: 21 estados T: 10,5 μseg @ 2 MHz.

Direccionamiento: Externo.

Banderas:



S	Z		H		P/V	N	C
?	1		?		?	1	

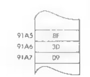
Ejemplo:

INIR

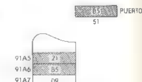
Antes:

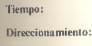






Después:





JP cc, pq

Función:

Formato:

Descripción:

Flujo de datos:

Tiempo:

Direccionamiento:

Códigos byte:

JP cc, pq

Salto condicional a la posición pq.

Función: Si cc es cierto: PC ← pq

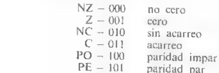
Formato:

| 1 | 1 | ←cc→ | 0 | 1 | 0 | byte 1

←q→ byte 2: dirección, byte inferior

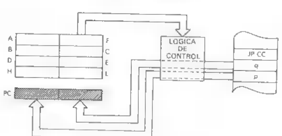
←p→ byte 3: dirección, byte superior

Descripción: Si se cumple la condición especificada, la dirección de dos bytes inmediatamente siguiente al código de operación se cargará en el contador del programa; el código de operación se carga en la parte inferior del PC. Si la condición no se satisface, se ignora la dirección; cc puede ser:



NZ – 000		no cero
Z – 001		cero
NC – 010		sin acarreo
C – 011		acarreo
PO – 100		paridad impar
PE – 101		paridad par
P – 110		más
M – 111		menos

Flujo de datos:



Tiempo: 3 ciclos M; 10 estados T; 5 μseg @ 2MHz.

Direccionamiento: Inmediato.

Códigos byte:

cc	NZ	Z	NC	C	PO	PE	P	M
	C2	CA	D2	DA	E2	EA	F2	FA

Banderas: S Z H P/V N C (efecto nulo)

Ejemplo: JP C, 3B24

Antes:

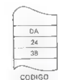
F: 51

PC: 0032

Después:

F: 51

PC: 3B24



JP pq

Función:

Formato:

Descripción:

Flujo de datos:

Tiempo:

Direccionamiento:

Banderas:

Ejemplo:



nulo)

espués:

51 F

3824

JP pq

Salto a la posición pq.

Función: PC ← pq

Formato:

| 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | byte 1: C3

q — byte 2: dirección, byte inferior

p — byte 3: dirección, byte superior

Descripción: El contenido de la posición de memoria inmediatamente siguiente al código de operación se carga en la mitad inferior del contador del programa y el de la posición siguiente a la anterior en la mitad superior del mismo contador. La siguiente instrucción se toma de esta nueva dirección.

Flujo de datos:

Tiempo: 3 ciclos M; 10 estados T; 5 μseg @ 2 MHz.

Direccionamiento: Inmediato.

Banderas:

S Z H P/V N C

(efecto nulo)

Ejemplo: JP 3025

Antes: PC 5520

Después: PC 3025

JP (HL)

Saltar a (HL).

Función: PC ← HL

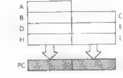
Formato:

1	1	1	0	1	0	0	1

E9

Descripción: El contenido del par de registros HL se carga en el cont del programa. La siguiente instrucción se toma de esta dirección.

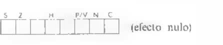
Flujo de datos:



Tiempo: 1 ciclo M; 4 estados T; 2 μseg @ 2 MHz.

Direccionamiento: Implícito.

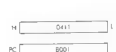
Banderas:



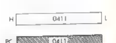
(efecto nulo)

Ejemplo: JP (HL)

Antes:



Después:



E9

CODIGO OBJETO

JP (IX)

Función:

Formato:

Descripción:

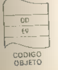
Flujo de datos:

Tiempo:

Direccionamiento:

Banderas:

Ejemplo:



JP (IX)

Salto a (IX).

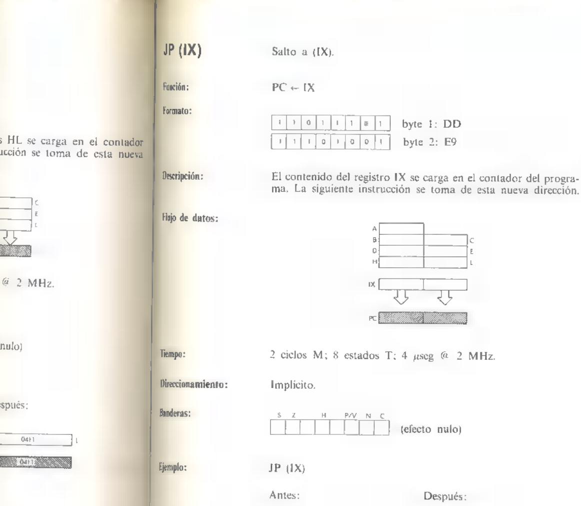
Función: PC ← IX

Formato:

1	1	0	1	1	1	0	1	byte 1: DD
1	1	1	0	1	0	0	1	byte 2: E9

Descripción: El contenido del registro IX se carga en el contador del programa. La siguiente instrucción se toma de esta nueva dirección.

Flujo de datos:



Tiempo: 2 ciclos M; 8 estados T; 4 μseg @ 2 MHz.

Direccionamiento: Implícito.

Banderas:

S	Z		H		P/V	N	C

(efecto nulo)

Ejemplo: JP (IX)

Antes:

IX 80F1

PC 384A

Después:

IX 80F1

PC 80F1

DD
E9
CODIGO OBJETO

JP (IY)

Salto a (IY).

Función: PC ← IY

Formato:

1	1	1	1	1	1	0	1	byte 1: FD
1	1	1	0	1	0	0	1	byte 2: E9

Descripción: El contenido del registro IY se lleva al contador del progra
La siguiente instrucción se toma de esta nueva dirección

Flujo de datos:

A
B C
D E
H L
IY
PC

Tiempo: 2 ciclos M; [illegible] estados T; 4 μseg @ 2 MHz.

Direccionamiento: Implícito.

Banderas:

S	Z		H		P/V	N	C

(efecto nulo).

Ejemplo: JP (IY)

Antes:

IY AA4B
PC E410

Después:

IY AA4B
PC AA4B

FD
E9
CODIGO OBJETO

JR cc, e

Función:

Formato:

Descripción:

Flujo de datos:

Tiempo:



JR cc, e

Salto relativo condicional de longitud e.

Función: Si cc es cierto, PC ← PC + e

Formato:

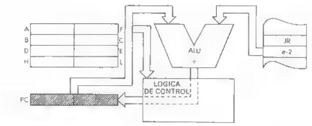
0	0	1	c	c	0	0	0	byte 1

← e-2 →	byte 2: valor del desplazamiento

Descripción: Si se satisface la condición especificada, el valor del desplazamiento dado se suma al contador del programa en aritmética de complemento a dos, para que el salto pueda darse hacia adelante o hacia atrás. El valor del desplazamiento se suma al valor de PC + 2 (después del salto), de manera que el desplazamiento efectivo es de −126 a +129 bytes. El ensamblador resta automáticamente 2 del valor del desplazamiento fuente para generar el código hexadecimal. Si la condición no se satisface, se ignora el valor del desplazamiento y la ejecución de instrucciones prosigue en secuencia; cc puede ser:

NZ – 00 NC – 10
Z – 01 C – 11

Flujo de datos:



Tiempo:

	Ciclos M	*Estados T*	*μseg @ 2 MHz*
condición cierta:	3	12	6
condición falsa:	2	7	3.5

Direccionamiento: Relativo.

Códigos byte:

cc:	NZ	Z	NC	C
	20	■	30	■

Banderas:

S	Z		■		P/V	N	C

(efecto nulo)

Ejemplo: JR NC, $ − 3 $ = PC actual

Antes:

F: 00

PC: B000

Después:

F: 00

PC: AFFD

CODIGO OBJETO
30
FB

JR e

Función:

Formato:

Descripción:

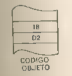
Flujo de datos:

Tiempo:

Direccionamiento:

Banderas:

Ejemplo:



JR e

Salto relativo de longitud e.

Función: PC ← PC + e

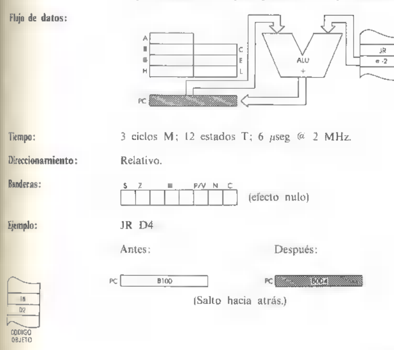
Formato:

0	0	0	1	1	0	0	0	byte 1: 18

← e-2 →	byte 2: valor del desplazamiento

Descripción: El valor del desplazamiento dado se suma al contador del programa en aritmética de complemento a 2, para que el salto pueda darse hacia adelante o hacia atrás. El valor del desplazamiento se suma al valor de PC + 2 (después del salto), de manera que el desplazamiento efectivo es de − 126 a + 129 bytes. El ensamblador resta automáticamente 2 del valor del desplazamiento fuente para generar el código hexadecimal.

Flujo de datos:



Tiempo: 3 ciclos M; 12 estados T; 6 μseg @ 2 MHz.

Direccionamiento: Relativo.

Banderas:

S	Z		H		P/V	N	C

(efecto nulo)

Ejemplo: JR D4

Antes: PC B100

Después: PC B0D4

(Salto hacia atrás.)

CODIGO OBJETO: 18, D2

LD dd, (nn)

Carga el par de registros dd a partir de las posiciones de memoria direccionadas por nn.

Función: $dd_{inf} \leftarrow (nn)$; $dd_{sup} \leftarrow (nn + 1)$

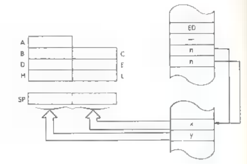
Formato:

1	1	1	0	1	1	0	1	byte 1: ED
0	1	d	d	1	0	1	1	byte 2
←			n				→	byte 3: dirección, byte inferior
←			n				→	byte 4: dirección, byte superior

Descripción: El contenido de la posición de memoria direccionada por la posición inmediatamente siguiente al código de operación se carga en la parte inferior del par de registros especificado. A continuación se carga el contenido de la posición de memoria siguiente a la anterior en la parte superior del mismo par de registros. El byte de orden inferior de la dirección nn sigue inmediatamente al código de operación; dd puede ser:

BC – 00	HL – 10
DE – 01	SP – 11

Flujo de datos:



Tiempo: 6 ciclos M; 20 estados T; 10 μseg @ 2 MHz

Direccionamiento: Directo.

Códigos byte:

Banderas:

Ejemplo:

ED
5B
21
50

CODIGO OBJETO

partir de las posiciones de

ED

dirección, byte inferior

dirección, byte superior

memoria direccionada por la
al código de operación se
de registros especificado. A
de la posición de memoria
superior del mismo par de
or de la dirección nn sigue
ración; dd puede ser:

HL – 10
SP – 11

@ 2 MHz

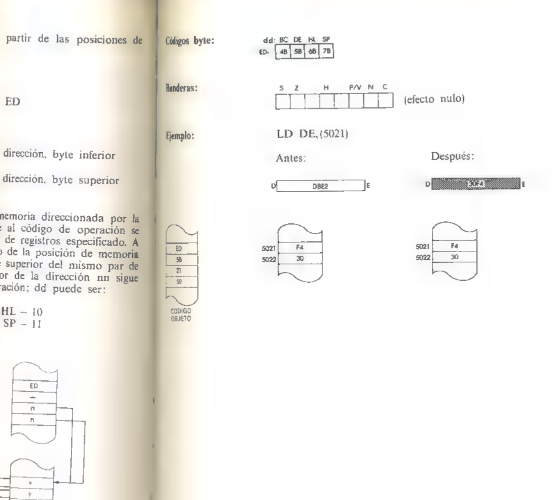
Códigos byte:

dd:	BC	DE	HL	SP
ED-	4B	5B	6B	7B

Banderas: S Z H P/V N C (efecto nulo)

Ejemplo: LD DE,(5021)

Antes: Después:

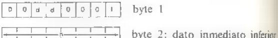


LD dd, nn

Carga el par de registros dd con el dato inmediato nn.

Función: dd ← nn

Formato:



byte 1

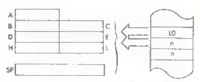
byte 2: dato inmediato inferior

byte 3: dato inmediato superior

Descripción: Los contenidos de las dos posiciones de memoria inmediata-mente siguientes al código de operación se cargan en el par de registros especificado. El byte de orden inferior del dato aparece justo a continuación del código de operación; dd puede ser

BC – 00 HL – 10
DE – 01 SP – 11

Flujo de datos:



Tiempo: 3 ciclos M; 10 estados T; 5 μseg @ 2 MHz.

Direccionamiento: Inmediato.

Códigos byte:

dd:	BC	DE	HL	SP
	01	11	21	31

Banderas:



(efecto nulo)

Ejemplo: LD DE, 4131

Antes: D [0394] E

Después: D [4131] E



LD r, n

Función:

Formato:

Descripción:

Flujo de datos:

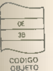
Tiempo:

Direccionamiento:

Códigos byte:

Banderas:

Ejemplo:



LD r, n

Carga el registro r con el dato inmediato n.

Función: r ← n

Formato:

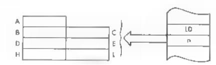
0	0	← r →	1	1	0	byte 1

← n →	byte 2: dato inmediato

Descripción: El contenido de la posición de memoria inmediatamente siguiente al código de operación se carga en el registro especificado; r puede ser:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Flujo de datos:



Tiempo: 2 ciclos M; 7 estados T; 3.5 μseg @ 2 MHz.

Direccionamiento: Inmediato.

Códigos byte:

r:	A	B	C	D	E	H	L
	3E	06	0E	16	1E	26	2E

Banderas:

S	Z		H		P/V	N	C

(efecto nulo)

Ejemplo: LD C, 3B

Antes: C 01

Después: C 3B



LD r, r'

Carga el registro r con el contenido del r'.

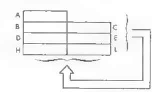
Función: r ← r'

Formato: | 0 | 1 | ← r → | ← r' → |

Descripción: El contenido del registro fuente especificado se carga en el registro destino especificado; r y r' pueden ser:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Flujo de datos:



Tiempo: 1 ciclo M; 4 estados T; 2 μseg @ 2 MHz.

Direccionamiento: Implícito.

Códigos byte:

(dest.) \ (fuente)	A	B	C	D	E	H	L
A	7F	78	79	7A	7B	7C	7D
B	47	40	41	42	43	44	45
C	4F	48	49	4A	4B	4C	4D
D	57	50	51	52	53	54	55
E	5F	58	59	5A	5B	5C	5D
H	67	60	61	62	63	64	65
L	6F	68	69	6A	6B	6C	6D

Banderas:



(efecto nulo)

Ejemplo: LD H, A

Antes:





Después:

A 8C



67
CODIGO OBJETO

LD (BC), A

Función:

Formato:

Descripción:

Flujo de datos:

Tiempo:

Direccionamiento:

Banderas:

Ejemplo:



LD (BC), A

Carga la posición de memoria indirectamente direccionada (BC) a partir del acumulador.

Función: (BC) ← A

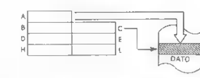
Formato:

0	0	0	0	0	0	1	0

02

Descripción: El contenido del acumulador se carga en la posición de memoria direccionada por el contenido del par de registros BC.

Flujo de datos:



Tiempo: 2 ciclos M; 7 estados T; 3.5 μseg @ 2 MHz.

Direccionamiento: Indirecto.

Banderas:



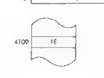
(efecto nulo)

Ejemplo: LD (BC), A

Antes:

Después:







LD (DE), A

Carga la posición de memoria indirectamente direccionada (DE) a partir del acumulador.

Función: (DE) ← A

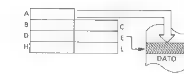
Formato:

0	0	0	1	0	0	1	0

12

Descripción: El contenido del acumulador se carga en la posición de memoria direccionada por el contenido del par de registros DE.

Flujo de datos:



Tiempo: 2 ciclos M; 7 estados T; 3.5 μseg @ 2 MHz.

Direccionamiento: Indirecto.

Banderas:

S	Z		H		P/V	N	C

(efecto nulo)

Ejemplo: LD (DE), A

Antes:

A ED

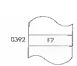
D 0392 E

Después:

A ED

D 0392 E







LD (HL), n

Función:

Formato:

Descripción:

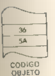
Flujo de datos:

Tiempo:

Direccionamiento:

Banderas:

Ejemplo:



LD (HL), n

Carga el dato inmediato n en la posición de memoria indirectamente direccionada (HL).

Función: (HL) ← n

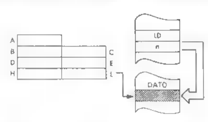
Formato:

0	0	1	1	0	1	1	0	byte 1: 36

← n →	byte 2: dato inmediato

Descripción: El contenido de la posición de memoria inmediatamente siguiente al código de operación se carga en la posición de memoria indirectamente direccionada por el apuntador HL.

Flujo de datos:



Tiempo: 3 ciclos M; 10 estados T; 5 μseg @ 2 MHz.

Direccionamiento: Inmediato/indirecto.

Banderas:

S	Z		H		P/V	N	C

(efecto nulo)

Ejemplo: LD (HL), 5A

Antes:

H [A342] L



Después:

H [A342] L

A342 [5A]



LD (HL), r

Carga la posición de memoria indirectamente direccionada (HL) a partir del registro r.

Función: (HL) ← r

Formato:

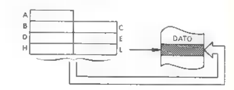
0	1	1	1	0	← r →

Descripción:

El contenido del registro especificado se carga en la posición de memoria direccionada por el par de registros HL; r puede ser

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Flujo de datos:



Tiempo: 2 ciclos M; 7 estados T; 3.5 μseg @ 2 MHz.

Direccionamiento: Indirecto.

Códigos byte:

r:	A	B	C	D	E	H	L
	77	70	71	72	73	■	75

Banderas:

S	Z		H		P/V	N	C

(efecto nulo)

Ejemplo: LD (HL), B

Antes:

B [81]

H [C501] L



Después:

B [81]

H [C501] L

C501 [81]

70

CODIGO OBJETO

LD r, (IX + d)

Función:

Formato:

Descripción:

Flujo de datos:

Tiempo:

Direccionamiento:

Códigos byte:

Banderas:

LD r, (IX + d)

Carga el registro r indirecto a partir de la posición de memoria indexada (IX + d).

Función: r ← (IX + d)

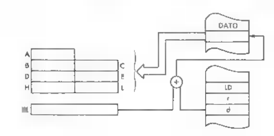
Formato:

1	1	0	1	1	1	0	1	byte 1: DD
0	1	←	r	→	1	1	0	byte 2
←			d				→	byte 3: valor del desplazamiento

Descripción: El contenido de la posición de memoria direccionada por el registro de índice IX más el valor del desplazamiento dado se carga en el registro especificado; r puede ser:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Flujo de datos:



Tiempo: 5 ciclos M; 19 estados T; 9.5 μseg @ 2 MHz.

Direccionamiento: Indexado.

Códigos byte:

r:	A	B	C	D	E	H	L	
DD-	7E	46	4E	56	5E	66	6E	-d

Banderas:

S	Z		H		P/V	N	C

(efecto nulo)

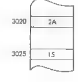
Ejemplo: LD E,(IX + 5)

Antes:

E: 03

IX: 3020

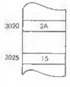
DD
5E
05
CODIGO OBJETO



Después:

E: 15

IX: 3020



LD r,(IY + d)

Función:

Formato:

Descripción:

Flujo de datos:

Tiempo:

Direccionamiento:

Códigos byte:

Banderas:

spués:

LD r, (IY + d)

Carga el registro indirecto r a partir de la posición de memoria indexada (IY + d).

Función: r ← (IY + d)

Formato:

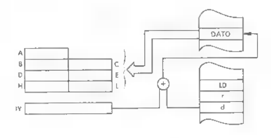
1	1	1	1	1	1	0	1	byte 1: FD
0	1	←	r	→	1	1	0	byte 2
←			d				→	byte 3: valor del desplazamiento

Descripción:

El contenido de la posición de memoria direccionada por el registro de índice IY más el valor del desplazamiento dado se carga en el registro especificado: r puede ser:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Flujo de datos:



Tiempo: 5 ciclos M; 19 estados T; 9.5 μseg @ 2 MHz.

Direccionamiento: Indexado.

Códigos byte:



Banderas:

S Z H P/V N C

(efecto nulo)

Ejemplo: LD A,(IY + 2)

Antes:

A E3

IY B005

FD
7E
02

CODIGO OBJETO

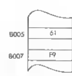
B005 61

B007 F9

Después:

A F9

IY B005



LD (IX + d), n

Función:

Formato:

Descripción:

Flujo de datos:

Tiempo:

Direccionamiento:

Banderas:

LD (IX + d), n

Carga la posición de memoria indexada (IX + d) con el dato inmediato n.

Función: (IX + d) ← n

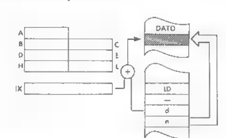
Formato:

1	1	0	1	1	1	0	1	byte 1: DD
0	0	1	1	0	1	1	0	byte 2: 36
←			d				→	byte 3: valor del desplazamiento
←			n				→	byte 4: dato inmediato

Descripción:

El contenido de la posición de memoria que sigue inmediatamente al valor del desplazamiento se transfiere a la posición de memoria direcccionada por el contenido del registro de índice más el valor del desplazamiento.

Flujo de datos:



Tiempo: 5 ciclos M; 19 estados T; 9.5 μseg @ 2 MHz.

Direccionamiento: Indexado/inmediato.

Banderas:

S	Z		H		P/V	N	C

(efecto nulo)

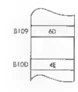
Ejemplo: LD (IX + 4), FF

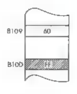
Antes: Después:

IX B109 IX B109

DD
36
04
FF

CODIGO OBJETO





LD (IY + d), n

Función:

Formato:

Descripción:

Flujo de datos:

Tiempo:

Direccionamiento:

Banderas:

LD (IY + d), n

Carga la posición de memoria indexada (IY + d) con el dato inmediato n.

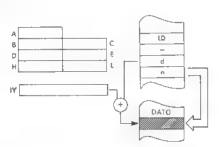
Función: (IY + d) ← n

Formato:

1	1	1	1	1	1	0	1	byte 1: FD
0	0	1	1	0	1	1	0	byte 2: 36
←			d				→	byte 3: valor del desplazamiento
←			n				→	byte 4: dato inmediato

Descripción: El contenido de la posición de memoria que sigue inmediatamente al valor del desplazamiento se transfiere a la posición de memoria direccionada por el registro de índice más el valor del desplazamiento.

Flujo de datos:



Tiempo: 5 ciclos M; 19 estados T; 9.5 μseg @ 2 MHz.

Direccionamiento: Indexado/inmediato.

Banderas:

S	Z		H		P/V	N	C

(efecto nulo)

Ejemplo: LD (IY + 3), BA

Antes:

IY 0100

	FD
	36
	03
	BA

CODIGO OBJETO

0100	D2
	62
	0F
0103	04

Después:

IY 0100

0100	D2
	62
	0F
0103	BA

LD (IX + d), r

Función:

Formato:

Descripción:

Flujo de datos:

Tiempo:

Direccionamiento:

Códigos byte:

Banderas:

LD (IX + d), r

Carga la posición de memoria indexada (IX + d) a partir del registro r.

Función: (IX + d) ← r

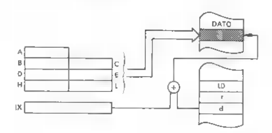
Formato:

1	1	0	1	1	1	0	1	byte 1: DD
0	1	1	1	0	←	r	→	byte 2
←			d				→	byte 3: valor del desplazamiento

Descripción: El contenido del registro especificado se carga en la posición de memoria direccionada por el contenido del registro de índice más el valor del desplazamiento dado; r puede ser:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Flujo de datos:



Tiempo: 5 ciclos M; 19 estados T; 9.5 μseg @ 2 MHz.

Direccionamiento: Indexado.

Códigos byte:



Banderas:

S	Z		H		P/V	N	C

(efecto nulo)

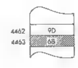
Ejemplo: LD (IX + 1), C

Antes: Después:

Antes		Después	
C	6B	C	6B
IX	4462	IX	4462







LD (IY + d), r

Función:

Formato:

Descripción:

Flujo de datos:

Tiempo:

Direccionamiento:

Códigos byte:

Banderas:

espués:



LD (IY + d), r

Carga la posición de memoria indexada (IY + d) a partir del registro r.

Función: (IY + d) ← r

Formato:

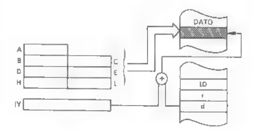
1	1	1	1	1	1	0	1	byte 1: FD
0	1	1	1	0	←	r	→	byte 2
←			d				→	byte 3: valor del desplazamiento

Descripción:

El contenido del registro especificado se carga en la posición de memoria direccionada por el contenido del registro de índice más el valor del desplazamiento dado; r puede ser:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Flujo de datos:



Tiempo: 5 ciclos M; 19 estados T; 9.5 µseg @ 2 MHz.

Direccionamiento: Indexado.

Códigos byte:

r:	A	B	C	D	E	H	L	
FD-	77	70	71	72	73	74	75	-d

Banderas:



(efecto nulo)

Ejemplo: LD (IY + 3), A

Antes:



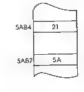
Después:

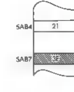
A 3E

IY 5AB4

FD
77
03

CODIGO OBJETO





LD A, (nn)

Ca

Función: A

Formato:

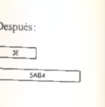
Descripción: E

Flujo de datos:

Tiempo:

Direccionamiento:

Banderas:



LD A, (nn)

Carga el acumulador a partir de la posición de memoria (nn).

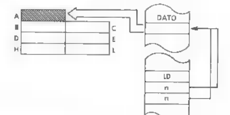
Función: A ← (nn)

Formato:

0	0	1	1	1	0	1	0	byte 1: 3A
n								byte 2: dirección, byte inferior
n								byte 3: dirección, byte superior

Descripción: El contenido de la posición de memoria direccionada por el contenido de las dos posiciones de memoria que siguen al código de operación se cargan en el acumulador. El byte de orden inferior de la dirección aparece justo a continuación del código de operación.

Flujo de datos:



Tiempo: 4 ciclos M; 13 estados T; 6.5 μseg @ 2 MHz.

Direccionamiento: Directo.

Banderas:

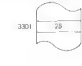
S	Z		H		P/V	N	C

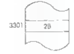
(efecto nulo)

Ejemplo: LD A,(3301)

Antes: Después:







LD (nn), A

Función:

Formato:

Descripción:

Flujo de datos:

Tiempo:

Direccionamiento:

Banderas:

LD (nn), A

Carga la posición de memoria directamente direccionada (nn) a partir del acumulador.

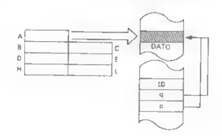
Función: (nn) ← A

Formato:

0	0	1	1	0	0	1	0	byte 1: 32
			n					byte 2: dirección, byte inferior
			n					byte 3: dirección, byte superior

Descripción: El contenido del acumulador se carga en la posición de memoria direccionada por el contenido de las posiciones de memoria que siguen al código de operación. El byte inferior de la dirección aparece justo a continuación del código de operación.

Flujo de datos:



Tiempo: 4 ciclos M; 13 estados T; 6.5 μseg @ 2 MHz.

Direccionamiento: Directo.

Banderas:

S	Z		H		P/V	N	C

(efecto nulo)



Ejemplo: LD (0321), A

Antes:

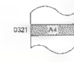
A A4

32
21
03
CODIGO OBJETO

0321 06

Después:

A A4



LD (nn), dd

Función:

Formato:

Descripción:

Flujo de datos:

Tiempo:

Direccionamiento:

Códigos byte:

LD (nn), dd

Carga las posiciones de memoria direccionadas por nn a partir del par de registros dd.

Función: $(nn) \leftarrow dd_{inf}$; $(nn + 1) \leftarrow dd_{sup}$

Formato:

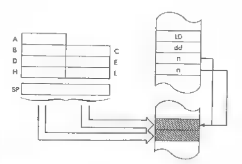
1	1	1	0	1	1	0	1	byte 1: ED
0	1	d	d	0	0	1	1	byte 2
←			n				→	byte 3: dirección, byte inferior
←			n				→	byte 4: dirección, byte superior

Descripción:

El contenido de orden inferior del par de registros especificado se carga en la posición de memoria direccionada por las posiciones de memoria que siguen al código de operación. El contenido de orden superior del par de registros se carga en la posición de memoria que sigue [redacted] la cargada [redacted] partir del orden inferior. El orden inferior de la dirección nn aparece justo [redacted] continuación del código de operación; dd puede ser:

BC – 00	HL – 10
DE – 01	SP – 11

Flujo de datos:



Tiempo: 6 ciclos M; 20 estados T; 10 μseg @ 2 MHz.

Direccionamiento: Directo.

Códigos byte:

dd:	BC	[redacted]	HL	SP
ED-	43	53	63	73

Banderas: S Z H P/V N C (efecto nulo)

Ejemplo: LD (040B), BC

Antes:

B 0221 C

040B	06
040C	AB

Después:

B 0221 C

040B	21
040C	02

ED
43
0B
04

CODIGO OBJETO

LD (nn), HL

Ca
de

Función: (nı

Formato:

Descripción: E d d c d c

Flujo de datos:

Tiempo:

Direccionamiento:

Banderas:

LD (nn), HL

Carga las posiciones de memoria direccionadas por nn a partir de HL.

Función:

(nn) ← L; (nn + 1) ← H

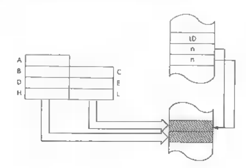
Formato:

0	0	1	0	0	0	1	0	byte 1: 22
n								byte 2: dirección, byte inferior
n								byte 3: dirección, byte superior

Descripción:

El contenido del registro L se carga en la posición de memoria direccionada por las posiciones de memoria que siguen al código de operación. El contenido del registro H se carga en la posición de memoria que sigue a la anterior. El byte inferior de la dirección nn aparece justo a continuación del código de operación.

Flujo de datos:



Tiempo:

5 ciclos M; 16 estados T; 8 μseg @ 2 MHz.

Direccionamiento:

Directo.

Banderas:



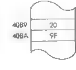
(efecto nulo)

Ejemplo: LD (40B9), HL



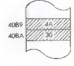
Antes:

H 304A L



Después:

H 304A L



22
B9
40

CODIGO OBJETO

LD (nn), IX

Función:

Formato:

Descripción:

Flujo de datos:

Tiempo:

Direccionamiento:

Banderas:

LD (nn), IX

Carga las posiciones de memoria direccionadas por nn a partir de IX.

Función:

$(nn) \leftarrow IX_{inf}$; $(nn + 1) \leftarrow IX_{sup}$

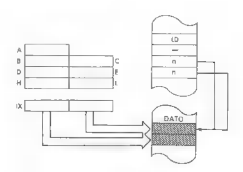
Formato:

1	1	0	1	1	1	0	1	byte 1: DD
0	0	1	0	0	0	1	0	byte 2: 22
n								byte 3: dirección, byte inferior
n								byte 4: dirección, byte superior

Descripción:

El contenido de orden inferior del registro IX se carga en la posición de memoria direccionada por el contenido de memoria que sigue al código de operación. El contenido de orden superior del registro IX se carga en la posición de memoria que sigue a la anterior. El byte inferior de la dirección nn aparece justo después del código de operación.

Flujo de datos:



Tiempo:

■ ciclos M; 20 estados T; 10 μseg @ 2 MHz.

Direccionamiento:

Directo.

Banderas:



(efecto nulo)

Ejemplo: LD (012B), IX

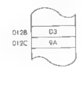


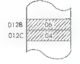


Antes: Después:









LD (nn), IY

Función:

Formato:

Descripción:

Flujo de datos:

Tiempo:

Direccionamiento:

Banderas:

LD (nn), IY

Carga las posiciones de memoria direccionadas por nn a partir de IY.

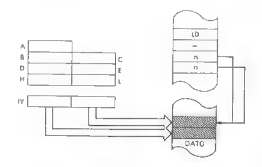
Función: $(nn) \leftarrow IY_{inf}$; $(nn + 1) \leftarrow IY_{sup}$

Formato:

1	1	1	1	1	1	0	1	byte 1: FD
0	0	1	0	0	0	1	0	byte 2: 22
n								byte 3: dirección, byte inferior
n								byte 4: dirección, byte superior

Descripción: El contenido de orden inferior del registro IY se carga en la posición de memoria direccionada por el contenido de la posición de memoria que sigue al código de operación. El contenido de orden superior de IY se carga en la posición de memoria que sigue a la anterior. El byte inferior de la dirección nn aparece justo después del código de operación.

Flujo de datos:



Tiempo: 6 ciclos M; 20 estados T; 10 μseg @ 2 MHz.

Direccionamiento: Directo.

Banderas:



(efecto nulo)

Ejemplo: LD (BD04), IY

Antes:

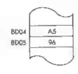
IY | D204

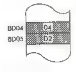
Después:

IY | D204

FD
22
04
BD

CODIGO OBJETO





LD A, (BC)

Función:

Formato:

Descripción:

Flujo de datos:

Tiempo:

Direccionamiento:

Banderas:

Ejemplo:



CODIGO OBJETO

LD A,(BC)

Carga el acumulador a partir de la posición de memoria indirectamente direccionada por el par de registros BC.

Función: A ← (BC)

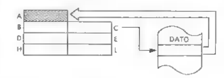
Formato:

0	0	0	0	1	0	1	0

0A

Descripción: El contenido de la posición de memoria direccionada por el contenido del par de registros BC se carga en el acumulador.

Flujo de datos:



Tiempo: 2 ciclos M; 7 estados T; 3.5 μseg. @ 2 MHz.

Direccionamiento: Indirecto.

Banderas:

S	Z		H		P/V	N	C

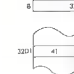
(efecto nulo)

Ejemplo: LD A,(BC)



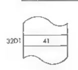
Antes:

A: AB
B: 32D1 C



Después:

A: 41
B: 32D1 C



LD A, (DE)

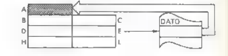
Carga el acumulador a partir de la posición de memoria indirectamente direccionada por el par de registros DE.

Función: A ← (DE)

Formato: 0 0 0 1 1 0 1 0   1A

Descripción: El contenido de la posición de memoria direccionada por el contenido del par de registros DE se carga en el acumulador.

Flujo de datos:



Tiempo: 2 ciclos M; 7 estados T; 3.5 μseg @ 2 MHz.

Direccionamiento: Indirecto.

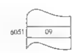
Banderas: S Z H P/V N C (efecto nulo)

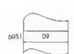
Ejemplo: LD A,(DE)

Antes: Después:









LD A, I

Función:

Formato:

Descripción:

Flujos de datos:

Tiempo:

Direccionamiento:

Banderas:

Ejemplo:



LD A, I

Carga el acumulador a partir del registro vector de interrupciones I.

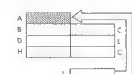
Función: A ← I

Formato:

1	1	1	0	1	1	0	1	byte 1: ED
0	1	0	1	0	1	1	1	byte 2: 57

Descripción: El contenido del registro vector de interrupciones I se carga en el acumulador.

Flujos de datos:



Tiempo: 2 ciclos M; 9 estados T; 4.5 μseg @ 2 MHz.

Direccionamiento: Implícito.

Banderas:



Se activa según el contenido de IFF2.

Ejemplo: LD A, I

Antes:

A 30 I 48

Después:



ED
57
CODIGO OBJETO

LD I, A

Carga el registro vector de interrupciones I a partir del acumulador.

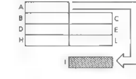
Función: I ← A

Formato:

1	1	1	0	1	1	0	1	byte 1: ED
0	1	0	0	0	1	1	1	byte 2: 47

Descripción: El contenido del acumulador se carga en el registro vector de interrupciones.

Flujo de datos:



Tiempo: 2 ciclos M; 9 estados T; 4.5 μseg @ 2 MHz.

Direccionamiento: Implícito.

Banderas:

S	Z		H		P/V	N	C

(efecto nulo)

Ejemplo: LD I, A

Antes: A 06 I D2

Después: A 06 I 06

ED
47
CODIGO OBJETO

LD A, R

Función:

Formato:

Descripción:

Flujo de datos:

Tiempo:

Direccionamiento:

Banderas:

Ejemplo:

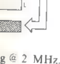


upciones I a partir del acumu-

: ED

: 47

carga en el registro vector de



g @ 2 MHz.

ulo)

spués:

LD A, R

Carga el acumulador a partir del registro de refresco de memoria R.

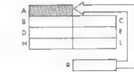
Función: A ← R

Formato:

1	1	1	0	1	1	0	1	byte 1: ED
0	1	0	1	1	1	1	1	byte 2: 5F

Descripción: El contenido del registro de refresco de memoria se carga en el acumulador.

Flujo de datos:



Tiempo: 2 ciclos M; 9 estados T; 4.5 μseg @ 2 MHz.

Direccionamiento: Implícito.

Banderas:

S	Z		H		P/V	N	C
●	●		0		×	0	

Se carga con el contenido de IFF2.

Ejemplo: LD A, R

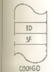


Antes: Después:









LD HL, (nn)

Carga el registro HL a partir de la posición de memoria direccionada por nn.

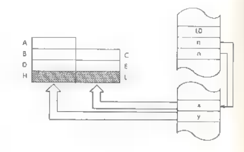
Función: L ← (nn); H ← (nn + 1)

Formato:

0	0	1	0	1	0	1	0	byte 1: 2A
n								byte 2: dirección, byte inferior
n								byte 3: dirección, byte superior

Descripción: El contenido de la posición de memoria direccionada por las posiciones de memoria que siguen al código de operación se cargan en el registro L. El contenido de la posición de memoria siguiente a la anterior se carga en el registro H. El byte inferior de la dirección nn aparece justo después del código de operación.

Flujo de datos:



Tiempo: 5 ciclos M; 16 estados T; 8 μseg @ 2 MHz.

Direccionamiento: Directo.

Banderas:

S	Z		H		P/V	N	C

(efecto nulo)

Ejemplo:

2A
24
00

CODIGO OBJETO

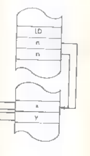
posición de memoria direc-

2A

dirección, byte inferior

dirección, byte superior

moria direccionada por las
al código de operacón se
de la posición de memoria
registro H. El byte inferior
spués del código de opera-



2 MHz.

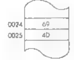
o)

Ejemplo: LD HL,(0024)

2A
24
00
CODIGO
OBJETO

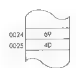
Antes:

H 08BF L



Después:





LD IX, nn

Carga el registro IX con el dato inmediato nn.

Función: IX ← nn

Formato:

| 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | byte 1: DD

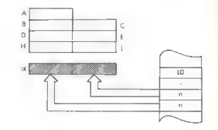
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | byte 2: 21

| ← n → | byte 3: dato inmediato, byte inferior

| ← n → | byte 4: dato inmediato, byte superior

Descripción: El contenido de la posición de memoria que sigue al código de operación se carga en el registro IX. El byte inferior aparece justo después del código de operación.

Flujo de datos:



Tiempo: 4 ciclos M; 14 estados T; 7 μseg @ 2 MHz.

Direccionamiento: Inmediato.

Banderas: S Z H P/V N C (efecto nulo)

Ejemplo: LD IX, B0B1

Antes: IX 306F

Después: IX B0B1

CODIGO OBJETO
DD
21
B1
B0

LD IX, (nn)

Función:

Formato:

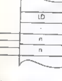
Descripción:

Flujo de datos:

Tiempo:

Direccionamiento:

Banderas:



LD IX, (nn)

Carga el registro IX a partir de las posiciones de memoria direcciondas por nn.

Función:

$IX_{inf} \leftarrow (nn);\ IX_{sup} \leftarrow (nn + 1)$

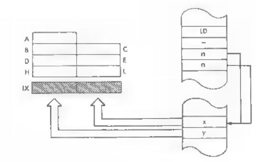
Formato:

1	1	0	1	1	1	0	1	byte 1: DD
0	0	1	0	1	0	1	0	byte 2: 2A
n								byte 3: dirección, byte inferior
n								byte 4: dirección, byte superior

Descripción:

El contenido de la posición de memoria direccionada por las posiciones de memoria que siguen al código de operación se carga en el byte inferior del registro IX. El contenido de la posición de memoria que sigue a la anterior se carga en el byte superior de IX. El byte inferior de la dirección nn aparece justo después del código de operación.

Flujo de datos:



Tiempo:

6 ciclos M; 20 estados T; 10 μseg @ 2 MHz.

Direccionamiento:

Directo.

Banderas:

S	Z		H		P/V	N	C

(efecto nulo)



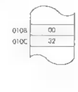
Ejemplo: LD IX,(010B)

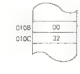
Antes: Después:

IX FF46 IX 3200

DD
2A
0B
01

CODIGO OBJETO





LD IY, nn

Función:

Formato:

Descripción:

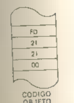
Flujo de datos:

Tiempo:

Direccionamiento:

Banderas:

Ejemplo:



LD IY, nn

Carga el registro IY con el dato inmediato nn.

Función: IY ← nn

Formato:

1	1	1	1	1	1	0	1	byte 1: FD
0	0	1	0	0	0	0	1	byte 2: 21
←			n				→	byte 3: dato inmediato, byte inferior
←			n				→	byte 4: dato inmediato, byte superior

Descripción: El contenido de la posición de memoria que sigue al código de operación se carga en el registro IY. El byte inferior aparece justo después del código de operación.

Flujo de datos:

A B C D E H L IY — LD, –, n, n

Tiempo: 4 ciclos M; 14 estados T; 7 μseg @ 2 MHz.

Direccionamiento: Inmediato.

Banderas:

S	Z		H		P/V	N	C

(efecto nulo)

Ejemplo: LD IY, 21

Antes: IY 0698

Después: IY 0021

CODIGO OBJETO: FD, 21, 21, 00



LD IY, (nn)

Carga el registro IY a partir de la posición de memoria direccionada por nn.

Función: $IY_{inf} \leftarrow (nn)$; $IY_{sup} \leftarrow (nn + 1)$

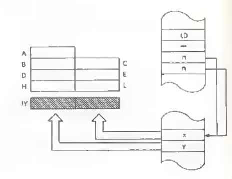
Formato:

1	1	1	1	1	1	0	1	byte 1: FD
0	0	1	0	1	0	1	0	byte 2: 2A

n	byte 3: dirección, byte inferior
n	byte 4: dirección, byte superior

Descripción: El contenido de la posición de memoria direccionada por las posiciones de memoria que siguen al código de operación se carga en el byte inferior del registro IY. El contenido de la posición de memoria que sigue a la anterior se carga en el byte superior del registro IY. El byte inferior de la dirección nn aparece justo después del código de operación.

Flujo de datos:



Tiempo: 6 ciclos M; 20 estados T; 10 μseg @ 2 MHz.

Direccionamiento: Directo.

Banderas:

S	Z		H		P/V	N	C

(efecto nulo)

posición de memoria direc-

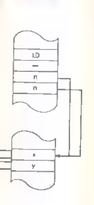
FD

2A

dirección, byte inferior

dirección, byte superior

moria direccionada por las
al código de operación se
ro IY. El contenido de la
anterior se carga en el byte
inferior de la dirección nn
le operación.



@ 2 MHz.

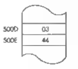
lo)

Ejemplo: LD IY, (500D)

Antes: IY 6002

Después: IY 4403





500D
500E
44

LD R, A

Carga el registro de refresco de memoria R a partir del acumulador.

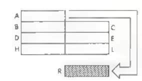
Función: R ← A

Formato:

1	1	1	0	1	1	0	1	byte 1: ED
0	1	0	0	1	1	1	1	byte 2: 4F

Descripción: El contenido del acumulador se carga en el registro de refresco de memoria.

Flujo de datos:



Tiempo: 2 ciclos M; 9 estados T; 4.5 μseg @ 2 MHz.

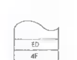
Direccionamiento: Implícito.

Banderas:

S	Z		H		P/V	N	C

(efecto nulo)

Ejemplo: LD R, A



Antes:



Después:



LD SP, HL

Función:

Formato:

Descripción:

Flujo de datos:

Tiempo:

Direccionamiento:

Banderas:

Ejemplo:



LD SP, HL

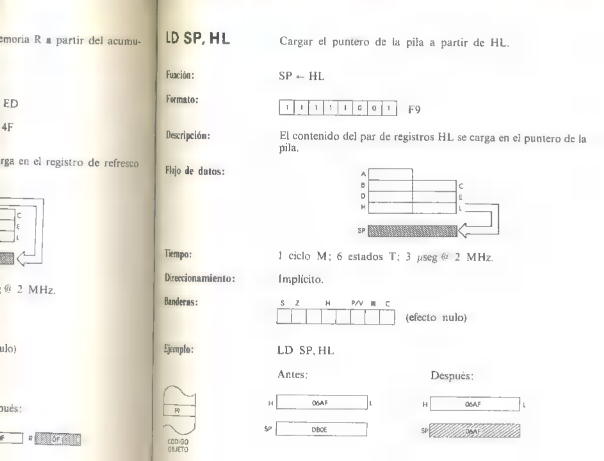
Cargar el puntero de la pila a partir de HL.

Función: SP ← HL

Formato: 1 1 1 1 1 0 0 1   F9

Descripción: El contenido del par de registros HL se carga en el puntero de la pila.

Flujo de datos:



Tiempo: 1 ciclo M; 6 estados T; 3 μseg @ 2 MHz.

Direccionamiento: Implícito.

Banderas: (efecto nulo)

Ejemplo: LD SP, HL

Antes:

Después:

LD SP, IX

Carga el puntero de la pila a partir del registro IX.

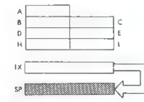
Función: SP ← IX

Formato:

1	1	0	1	1	1	0	1	byte 1: DD
1	1	1	1	1	0	0	1	byte 2: F9

Descripción: El contenido del registro IX se carga en el puntero de la pila.

Flujo de datos:



Tiempo: 2 ciclos M; 10 estados T; 5 μseg @ 2 MHz.

Direccionamiento: Implícito.

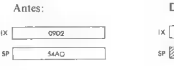
Banderas:

S Z H P/V N C

(efecto nulo)

Ejemplo: LD SP, IX

Antes:



Después:

IX 09D2

SP 09D2



DD

F9

OBJETO
CODIGO

LD SP, IY

Función:

Formato:

Descripción:

Flujo de datos:

Tiempo:

Direccionamiento:

Banderas:

Ejemplo:



LD SP, IY

Carga el puntero de la pila a partir del registro IY.

Función: SP ← IY

Formato:

1	1	1	1	1	1	0	1	byte 1: FD
1	1	1	1	1	0	0	1	byte 2: F9

Descripción: El contenido del registro IY se carga en el puntero de la pila.

Flujo ■ datos:

Tiempo: 2 ciclos M; 10 estados T; 5 μseg @ 2 MHz.

Direccionamiento: Implícito.

Banderas:

S	Z		H		P/V	N	C

(efecto nulo)

Ejemplo: LD SP, IY

Antes:

Después:

LDD

Carga de bloque con decremento.

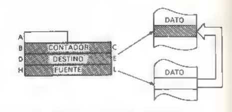
Función: (DE) ← (HL); DE ← DE − 1; HL ← HL − 1; BC ← BC − 1

Formato:

1	1	1	0	1	1	0	1	byte 1: ED
1	0	1	0	1	0	0	0	byte 2: A8

Descripción: El contenido de la posición de memoria direccionada por HL se carga en la posición de memoria direccionada por DE. A continuación se decrementan BC, DE y HL.

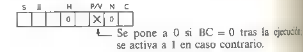
Flujo de datos:



Tiempo: 4 ciclos M; 16 estados T; 8 μseg @ 2 MHz.

Direccionamiento: Indirecto.

Banderas:



Se pone a 0 si BC = 0 tras la ejecución; se activa a 1 en caso contrario.

Ejemplo:

ED
A8
CODIGO OBJETO

← HL − 1; BC ← BC − 1

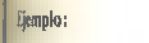
ED

A8

oria direccionada por HL se
reccionada por DE. A conti-
y HL.

@ 2 MHz.

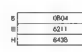
i BC = 0 tras la ejecución;
caso contrario.

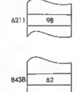


LDD



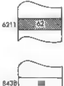
Antes:





Después:





LDDR

Carga de bloque con decremento repetida.

Función:

(DE) ← (HL); DE ← DE − 1; HL ← HL − 1;
BC ← BC − 1; repetir hasta que BC = 0

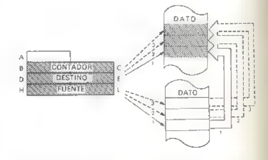
Formato:

1	1	1	0	1	1	0	1	byte 1: ED
1	0	1	1	1	0	0	0	byte 2: B8

Descripción:

El contenido de la posición de memoria direccionada por HL se carga en la posición de memoria direccionada por DE. A continuación se decrementan DE, HL y BC. Si BC ≠ 0, el contador del programa se decrementa en 2, y la instrucción vuelve a ejecutarse.

Flujo de datos:



Tiempo:

BC ≠ 0: 5 ciclos M; 21 estados T; 10.5 μseg @ 2 MHz.
BC = 0: 4 ciclos M; 16 estados T; 8 μseg @ 2 MHz.

Direccionamiento: Indirecto.

Banderas:

S	Z		H		P/V	N	C
			0		0	0	

repetida.

← HL − 1;
BC = 0

ED

B8

noria direccionada por HL se
ireccionada por DE. A conti-
BC. Si BC ≠ 0, el contador
2, y la instrucción vuelve a

DATO

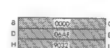
DATO

T; 10.5 μseg @ 2 MHz.
T; 8 μseg @ 2 MHz.

Ejemplo: LDDR

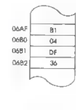
Antes: Después:

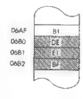


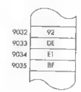


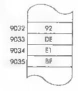


CODIGO OBJETO









LDI

Carga de bloque con incremento.

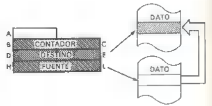
Función: (DE) ← (HL); DE ← DE + 1; HL ← HL + 1; BC ← BC − 1

Formato:

1	1	1	0	1	1	0	1	byte 1: ED
1	0	1	0	0	0	[illegible]	0	byte 2: A0

Descripción: El contenido de la posición de memoria direccionada por HL se carga en la posición de memoria direccionada por DE. A continuación se incrementan DE y HL y se decrementa el par de registros BC.

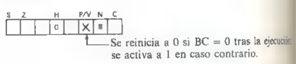
Flujo de datos:



Tiempo: 4 ciclos M; 16 estados T; [illegible] μseg [illegible] 2 MHz.

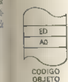
Direccionamiento: Indirecto.

Banderas:



Se reinicia a 0 si BC = 0 tras la ejecución; se activa a 1 en caso contrario.

Ejemplo:

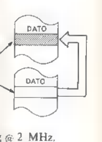


← HL + 1; BC ← BC − 1

ED

A0

moria direccionada por HL se
ireccionada por DE. A conti-
.. y se decrementa el par de

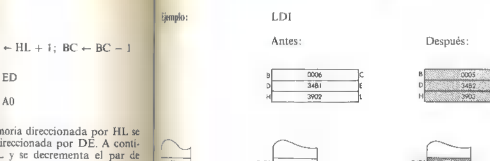


@ 2 MHz.

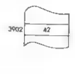
0 si BC = 0 tras la ejecución;
en caso contrario.

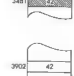
Ejemplo: LDI

Antes: Después:









LDIR

Carga de bloque con incremento repetida.

Función:

(DE) ← (HL); DE ← DE + 1; HL ← HL + 1;
BC ← BC − 1; repetir hasta que BC = 0

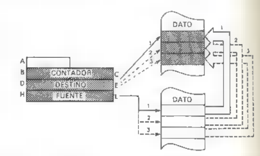
Formato:

1	1	1	0	1	1	0	1	byte 1: ED
1	0	1	1	0	0	0	0	byte 2: B0

Descripción:

El contenido de la posición de memoria direccionada por HL se carga en la posición de memoria direccionada por DE. A conti- nuación se incrementan DE y HL y se decrementa BC. Si BC ≠ 0, el contador del programa se decrementa en 2, y la instrucción vuelve a ejecutarse.

Flujo de datos:



Tiempo:

Si BC ≠ 0: 5 ciclos M; 21 estados T; 10.5 μseg @ 2 MHz
Si BC = 0: 4 ciclos M; 16 estados T; 8 μseg @ 2 MHz

Direccionamiento: Indirecto.

Banderas:

S	Z		H		P/V	N	C
			0		0	0	

Ejemplo:



4A03
4A04
4A05

962A
962B
962C

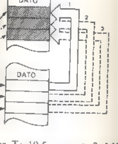
o repetida.

L ← HL + 1;
: BC = 0

: ED

B0

moria direccionada por HL se
direccionada por DE. A conti-
HL y se decrementa BC. Si
na se decrementa en 2, y la

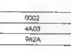


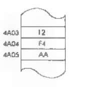
os T; 10.5 μseg @ 2 MHz.
os T; 8 μseg @ 2 MHz.

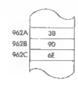
Ejemplo: ED B0 CODIGO OBJETO

LDIR

Antes:



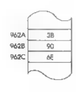




Después:

B 0000 C
D 4A05 E
H 962C L

4A03 3B
4A04 90
4A05 AA



LD r, (HL)

Carga el registro r indirecto a partir de la posición de memoria (HL).

Función: r ← (HL)

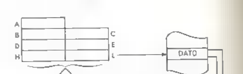
Formato:

0	1	← r →	1	1	0

Descripción: El contenido de la posición de memoria direccionada por HL se carga en el registro especificado; r puede ser:

A – 111   E – 011
B – 000   H – 100
C – 001   L – 101
D – 010

Flujo de datos:



Tiempo: 2 ciclos M; 7 estados T; 3.5 μseg @ 2 MHz.

Direccionamiento: Indirecto.

Códigos byte:



Banderas:

S	Z		H		P/V	N	C

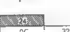
(efecto nulo)

Ejemplo: LD D,(HL)

Antes: Después:

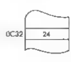
D 3A
H 0C   32 L

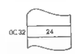




56

CODIGO OBJETO





NEG

Función:

Formato:

Descripción:

Flujo de datos:

Tiempo:

Direccionamiento:

Banderas:

Ejemplo:



NEG

Negativiza el acumulador.

Función: A ← 0 − A

Formato:

1	1	1	0	1	1	0	1	byte 1: ED
0	1	0	0	0	1	0	0	byte 2: 44

Descripción: El contenido del acumulador se resta de cero (en complemento a dos), y el resultado vuelve a almacenarse en el acumulador.

Flujo de datos:

A B C D E H L ALU

Tiempo: 2 ciclos M; 8 estados T; 4 μseg @ 2 MHz.

Direccionamiento: Implícito.

Banderas:

S	Z		H		P/V	N	C
●	●		●		●	1	●

C se activa a 1 si A era 0 antes de la instrucción.
P se activa a 1 si A era 80H.

Ejemplo: NEG

Antes: A 32

Después: A CE

ED
44
CODIGO OBJETO

NOP

No opera.

Función: Retardo.

Formato:

0	0	0	0	0	0	0	0	00

Descripción: No ocurre nada durante un ciclo M.

Flujo de datos: No actúa.

A			
B			C
D			E
H			L

Tiempo: 1 ciclo M; 4 estados T; 2 μseg @ 2 MHz.

Direccionamiento: Implícito.

Banderas:

S	Z		H		P/V	N	C

(efecto nulo)

OR s

Función:

Formato:

(HI

(IX +

(IY +

Descripción:

Flujo de datos:

o M.

o actúa.

C

E

L

a 2 MHz.

ulo)

OR s

O lógica al acumulador y el operando s.

Función: A ← A ∨ s

Formato: s puede ser r, n, (HL), (IX + d) o (IY + d).

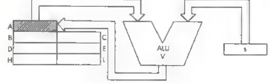
r	1	0	1	1	0	← r →			
n	1	1	1	1	0	1	1	0	byte 1: F6
	← n →								byte 2: dato inmediato
(HL)	1	0	1	1	0	1	1	0	byte 1: B6
(IX + d)	1	1	0	1	1	1	0	1	byte 1: DD
	1	0	1	1	0	1	1	0	byte 2: B6
	← d →								byte 3: valor del desplazamiento
(IY + d)	1	1	1	1	1	1	0	1	byte 1: FD
	1	0	1	1	0	1	1	0	byte 2: B6
	← d →								byte 3: valor del desplazamiento

r puede ser:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Descripción: El acumulador y el operando especificado se someten a la operación O lógica, y el resultado se almacena en el acumulador; s se define en la descripción de instrucciones ADD similares.

Flujo de datos:



Tiempo:

s	*Ciclos M*	*Estados T*	*μseg @ 2 MHz*
r	1	4	4
n	2	7	3.5
(HL)	2	7	3.5
(IX + d)	5	19	9.5
(IY + d)	5	19	9.5

Direccionamiento: r: implícito; n: inmediato; (HL): indirecto; (IX + d), (IY + d) indexado.

Códigos byte: OR r

r:	A	B	C	D	E	H	L
	B7	B0	B1	B2	B3	B4	B5

Banderas:

S	Z		H		P/V	N	C
●	●		0		●	0	0

Ejemplo: OR B

Antes:

A 06
B B9

Después:





OTDR

Función:

Formato:

Descripción:

Flujo de datos:

Tiempo:

Direccionamiento:

Banderas:

Estados T	µseg @ 2 MHz
4	4
7	3.5
7	3.5
19	9.5
19	9.5

directo; (IX + d), (IY + d):



ués:

OTDR

Salida de bloque con decremento.

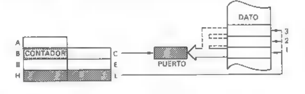
Función: (C) ← (HL); B ← B − 1; HL ← HL − 1; repetir hasta que B = 0.

Formato:

1	1	1	0	1	1	0	1	byte 1: ED
1	0	1	1	1	0	1	1	byte 2: BB

Descripción: El contenido de la posición de memoria direccionada por el par de registros HL se lleva a la salida del dispositivo periférico direccionado por el contenido del registro C. Tanto el registro B como el par HL se decrementan. Si B ≠ 0, el contador del programa se decrementa en 2, y la instrucción vuelve a ejecutarse. C proporciona los bits A0 a A7 del *bus* de direcciones; B proporciona, tras decremento, los bits A8 a A15.

Flujo de datos:



Tiempo: B = 0: 4 ciclos M; 16 estados T; 8 µseg @ 2 MHz.
B ≠ 0: 5 ciclos M; 21 estados T; 10.5 µseg @ 2 MHz.

Direccionamiento: Externo.

Banderas:

S	Z		H		P/V	N	C
?	1		?		?	1	

Ejemplo: OTDR



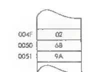
Antes:



Después:







OTIR

Función:

Formato:

Descripción:

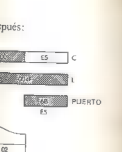
Flujo de datos:

Tiempo:

Direccionamiento:

Banderas:

pués:



OTIR

Salida de bloque con incremento.

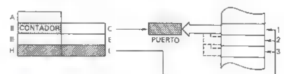
Función: (C) ← (HL); B ← B − 1; HL ← HL + 1; repetir hasta que B = 0

Formato:

1	1	1	0	1	1	0	1	byte 1: ED
1	0	1	1	0	0	1	1	byte 2: B3

Descripción: El contenido de la posición de memoria direccionada por el par de registros HL se lleva a la salida del dispositivo periférico direccionado por el contenido del registro C. El registro B se decrementa y el par HL se incrementa. Si B ≠ 0, el contador del programa se decrementa en 2, y la instrucción vuelve a ejecutarse. C proporciona los bits A0 a A7; B proporciona, tras decremento, los bits A8 a A15.

Flujo de datos:



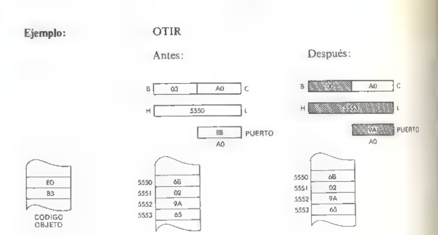
Tiempo: B = 0: 4 ciclos M; 16 estados T; 8 μseg @ 2 MHz.
B ≠ 0: 5 ciclos M; 21 estados T; 10.5 μseg @ 2 MHz.

Direccionamiento: Externo.

Banderas:

S	Z		H		P/V	N	C
?	1		?		?	1	

Ejemplo: OTIR



OUT (C), r

Función:

Formato:

Descripción:

Flujo de datos:

Tiempo:

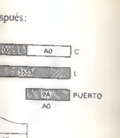
Direccionamiento:

Banderas:

Códigos byte:

Ejemplo:





OUT (C), r

Salida del registro r al puerto C.

Función: (C) ← r

Formato:

1	1	1	0	1	1	0	1	byte 1: ED
0	1	←	r	→	0	0	1	byte 2

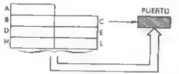
Descripción:

El contenido del registro especificado se lleva al dispositivo periférico direccionado por el contenido del registro C; r puede ser:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

El registro C proporciona los bits A0 a A7 del *bus* de direcciones; el registro B proporciona los bits A8 a A15.

Flujo de datos:



Tiempo: 3 ciclos M; 12 estados T; 6 μseg @ 2 MHz.

Direccionamiento: Externo.

Banderas:

S	Z		H		P/V	N	C

(efecto nulo)

Códigos byte:

r:	A	B	C	D	E	H	L
ED-	79	41	49	51	59	[redacted]	69

Ejemplo:

OUT (C), B

Antes:

B 09 | F1 C

PUERTO 88 (F1)

Después:

B 09 | F1 C

PUERTO 09 (F1)

ED
41

CODIGO OBJETO

OUT (N), A

Salida del acumulador al puerto N.

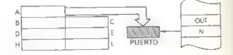
Función: (N) ← A

Formato:

1	1	0	1	0	0	1	1	byte 1: D3
←			N				→	byte 2: dirección puerto

Descripción: El contenido del acumulador se lleva al dispositivo periférico direccionado por el contenido de la posición de memoria que sigue al código de operación.

Flujo de datos:



Tiempo: 3 ciclos M; 11 estados T; 5.5 μseg @ 2 MHz.

Direccionamiento: Externo.

Banderas:

S	Z		H		P/V	N	C

(efecto nulo)

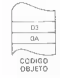
Ejemplo: OUT (0A), A

Antes:

A 51 · FF PUERTO 0A

Después:

A 51 · 51 PUERTO 0A



OUTD

Función:

Formato:

Descripción:

Flujo de datos:

Tiempo:

Direccionamiento:

Banderas:

Ejemplo:



N.

D3

dirección puerto

a al dispositivo periférico di-
osición de memoria que sigue



g @ 2 MHz.

ulo)

pués:

51 PUERTO
0A

OUTD

Salida con decremento.

Función: (C) ← (HL); BC ← B − 1; HL ← HL − 1

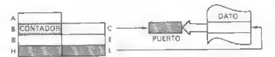
Formato:

1	1	1	0	1	1	0	1	byte 1: ED
1	0	1	0	1	0	1	1	byte 2: AB

Descripción: El contenido de la posición de memoria direccionada por el par de registros HL se lleva al dispositivo periférico direccionado por el contenido del registro C. El registro B y el par HL se decrementan. C proporciona los bits A0 a A7 del *bus* de direcciones; B proporciona, tras decremento, los bits A8 a A15.

Flujo de datos:





Tiempo: 4 ciclos M; 16 estados T; 8 μseg @ 2 MHz.

Direccionamiento: Externo.

Banderas:

S	Z		H		P/V	N	C
?	X		?		?	1	

Se activa a 1 si B = 0 tras la ejecución; en caso contrario se reinicia a 0.

Ejemplo: OUTD

Antes:

B 30 9A C
H 22BF L
06 PUERTO
9A

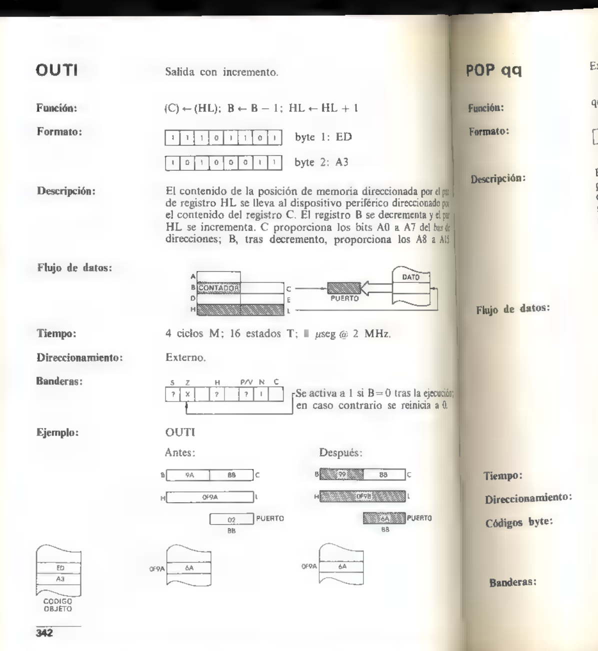
22BF 4A

Después:

B 2F 9A C
H 22BE L
4A PUERTO

22BF 4A

ED
AB
CODIGO OBJETO



OUTI

Salida con incremento.

Función: (C) ← (HL); B ← B − 1; HL ← HL + 1

Formato:

1	1	1	0	1	1	0	1	byte 1: ED
1	0	1	0	0	0	1	1	byte 2: A3

Descripción: El contenido de la posición de memoria direccionada por el par de registro HL se lleva al dispositivo periférico direccionado por el contenido del registro C. El registro B se decrementa y el par HL se incrementa. C proporciona los bits A0 a A7 del bus de direcciones; B, tras decremento, proporciona los A8 a A15.

Flujo de datos:

Tiempo: 4 ciclos M; 16 estados T; [illegible] μseg @ 2 MHz.

Direccionamiento: Externo.

Banderas:

Ejemplo: OUTI

POP qq

Ext

Función: qq

Formato:

Descripción:

Flujo de datos:

Tiempo:

Direccionamiento:

Códigos byte:

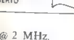
Banderas:

. + 1

ED

A3

noria direccionada por el par
o periférico direccionado por
tro B se decrementa y el par
los bits A0 a A7 del *bus* de
proporciona los A8 a A15.



@ 2 MHz.

a 1 si B=0 tras la ejecución;
contrario se reinicia a 0.

ués:

BB C

0F9B L

6A PUERTO

POP qq

Extrae de la pila el par de registros qq.

Función:

$qq_{inf} \leftarrow (SP)$; $qq_{sup} \leftarrow (SP+1)$; $SP \leftarrow SP + 2$

Formato:

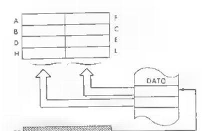
1	1	q	q	0	0	0	1

Descripción:

El contenido de la posición de memoria direccionada por el puntero de la pila se carga en el byte inferior del par de registros especificado, y a continuación se incrementa el puntero. El contenido de la posición de memoria que ahora direcciona el puntero se carga en el byte superior del par de registros, y el puntero vuelve a incrementarse; qq puede ser:

BC – 00 HL – 10
DE – 01 AF – 11

Flujo de datos:



Tiempo: 3 ciclos M; 10 estados T; 5 μseg @ 2 MHz.

Direccionamiento: Indirecto.

Códigos byte:

qq:	BC	[illegible]	HL	AF
	C1	D1	E1	F1

Banderas:

S	Z		H		P/V	N	C

(efecto nulo)



Ejemplo:



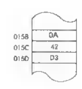
POP BC

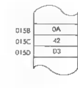
Antes: Después:

B [B90A] C — SP [015B]

B [420A] C — SP [015D]







POP IX

Función:

Formato:

Descripción:

Flujo de datos:

Tiempo:

Direccionamiento:

Banderas:

POP IX

Extrae de la pila el registro IX.

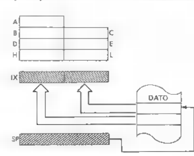
Función: $IX_{inf} \leftarrow (SP)$; $IX_{sup} \leftarrow (SP + 1)$; $SP \leftarrow SP + 2$

Formato:

1	1	0	1	1	1	0	1	byte 1: DD
1	1	1	0	0	0	0	1	byte 2: E1

Descripción: El contenido de la posición de memoria direccionada por el puntero de la pila se carga en el byte inferior del registro IX, y se incrementa el puntero. El contenido de la posición de memoria que ahora direcciona el puntero se carga en el byte superior del registro IX, y el puntero vuelve a incrementarse.

Flujo de datos:



Tiempo: 4 ciclos M; 14 estados T; 7 μseg @ 2 MHz.

Direccionamiento: Indirecto.

Banderas:

S	Z		H		P/V	N	C

(efecto nulo)

Ejemplo: POP IX

Antes:

IX 0001

SP 090B

Después:

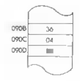
IX 0436

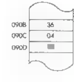
SP 090D

DD

E1

CODIGO OBJETO





POP IY

Función:

Formato:

Descripción:

Flujo de datos:

Tiempo:

Direccionamiento:

Banderas:

POP IY

Extrae de la pila el registro IY.

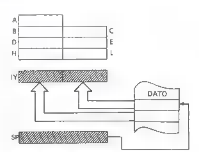
Función: $IY_{inf} \leftarrow (SP);\ IY_{sup} \leftarrow (SP + 1);\ SP \leftarrow SP + 2$

Formato:

1	1	1	1	1	1	0	1	byte 1: FD
1	1	1	0	0	0	0	1	byte 2: E1

Descripción: El contenido de la posición de memoria direccionada por el puntero de la pila se carga en el byte de orden inferior del registro IY, y a continuación se incrementa el puntero. El contenido de la posición de memoria que ahora direcciona el puntero se carga en el byte superior del registro IY, y el puntero vuelve a incrementarse.

Flujo de datos:



Tiempo: 4 ciclos M; 14 estados T; 7 μseg @ 2 MHz.

Direccionamiento: Indirecto.

Banderas:

S	Z		H		P/V	N	C	
								(efecto nulo)

Ejemplo: POP IY

Antes:

IY 032A

SP 3004

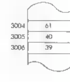
Después:

IY 4061

SP 3006

FD
E1

CODIGO OBJETO



3004 61
3005 40
3006 39

PUSH qq

Función:

Formato:

Descripción:

Flujo de datos:

Tiempo:

Direccionamiento:

Códigos byte:

Banderas:

PUSH qq

Introduce el par de registros qq en la pila.

Función:

$(SP - 1) \leftarrow qq_{sup}$; $(SP - 2) \leftarrow qq_{inf}$; $SP \leftarrow SP - 2$

Formato:

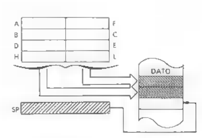
1	1	q	q	0	1	0	1

Descripción:

Se decrementa el puntero de la pila, y a continuación se carga el contenido del byte superior del par de registros especificado en la posición de memoria direccionada por el puntero de la pila. Este vuelve a decrementarse, y se carga el byte inferior del par de registros en la posición direccionada ahora por el puntero; qq puede ser:

BC – 00	HL – 10
DE – 01	AF – 11

Flujo de datos:



Tiempo:

3 ciclos M; 11 estados T; 6.5 μseg @ 2 MHz.

Direccionamiento:

Indirecto.

Códigos byte:

qq:	BC	DE	HL	AF
	C5	D5	E5	F5

Banderas:

S	Z		H		P/V	N	C

(efecto nulo)

Ejemplo: PUSH DE

Antes:

D 0A03 E

SP 00B1

00AF	B6
00B0	9A
00B1	0F

Después:

D 0A03 E

SP 00AF

00AF	03
00B0	0A
00B1	0F

D5

CODIGO OBJETO

PUSH IX

Función:

Formato:

Descripción:

Flujo de datos:

Tiempo:

Direccionamiento:

Banderas:

PUSH IX

Introduce IX en la pila.

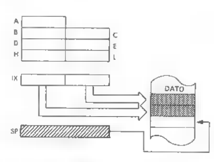
Función: $(SP - 1) \leftarrow IX_{sup}$; $(SP - 2) \leftarrow IX_{inf}$; $SP \leftarrow SP - 2$

Formato:

1	1	0	1	1	1	0	1	byte 1: DD
1	1	1	0	0	1	0	1	byte 2: E5

Descripción: Se decrementa el puntero de la pila y se carga el byte superior del registro IX en la posición de memoria direccionada por el puntero. Este vuelve a decrementarse, y el byte inferior de IX se carga en la posición de memoria direccionada ahora por el puntero.

Flujo de datos:



Tiempo: 4 ciclos M; 15 estados T; 7.5 μseg @ 2 MHz.

Direccionamiento: Indirecto.

Banderas:



(efecto nulo)



Ejemplo: PUSH IX

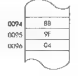
Antes:

IX 04A2

SP 0096

DD
E5

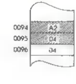
CODIGO OBJETO



Después:

IX 04A2

SP 0094



PUSH IY

Función:

Formato:

Descripción:

Flujo de datos:

Tiempo:

Direccionamiento:

Banderas:

PUSH IY

Introduce IY en la pila.

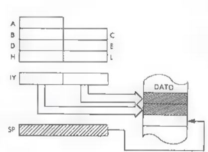
Función: $(SP-1) \leftarrow IY_{sup}$; $(SP-2) \leftarrow IY_{inf}$; $SP \leftarrow SP-2$

Formato:

1	1	1	1	1	1	0	1	byte 1: FD
1	1	1	0	0	1	0	1	byte 2: E5

Descripción: Se decrementa el puntero de la pila y se carga el byte superior del registro IY en la posición de memoria direccionada por el puntero. Este vuelve a decrementarse, y el byte inferior de IY se carga en la posición de memoria direccionada ahora por el puntero.

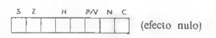
Flujo de datos:



Tiempo: 3 ciclos M; 15 estados T; 7.5 μseg @ 2 MHz.

Direccionamiento: Indirecto.

Banderas:

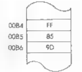


(efecto nulo)

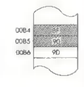
Ejemplo: PUSH IY

Antes:





Después:



RES b, s

Función:

Formato:

r

(HL)

(IX + d)

(IY + d)

Descripción:

RES b, s

Poner a 0 el bit b del operando s.

Función: $s_b \leftarrow 0$

Formato: s:

Operando	Bits	Byte
r	1 1 0 0 1 0 1 1	byte 1: CB
	1 0 ←b→ ←r→	byte 2
(HL)	1 1 0 0 1 0 1 1	byte 1: CB
	1 0 ←b→ 1 1 0	byte 2
(IX + d)	1 1 0 1 1 1 0 1	byte 1: DD
	1 1 0 0 1 0 1 1	byte 2: CB
	←d→	byte 3: valor del desplazamiento
	1 0 ←b→ 1 1 0	byte 4
(IY + d)	1 1 1 1 1 1 0 1	byte 1: FD
	1 1 0 0 1 0 1 1	byte 2: CB
	←d→	byte 3: valor del desplazamiento
	1 0 ←b→ 1 1 0	byte 4

b puede ser:

0 – 000	4 – 100
1 – 001	5 – 101
2 – 010	6 – 110
3 – 011	7 – 111

r puede ser:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Descripción: El bit especificado de la posición determinada por s se pone a 0; s se define en la descripción de instrucciones BIT similares.

Flujo de datos:

Tiempo:

s	*Ciclos M*	*Estados T*	*μseg @ 2 MHz*
r	2	8	4
(HL)	4	15	7.5
(IX + d)	6	23	11.5
(IY + d)	6	23	11.5

Direccionamiento: r: implicito; (HL): indirecto; (IX + d), (IY + d): indexado.

Códigos byte: RES b, r

CB—

b: \ r:	A	B	C	D	E	H	L
0	87	80	81	[illegible]	83	[illegible]	[illegible]
1	8F	88	89	8A	8B	8C	8D
2	97	90	91	92	93	94	[illegible]
3	9F	98	99	9A	9B	9C	9D
4	A7	A0	A1	A2	A3	A4	A5
5	AF	A8	A9	AA	AB	AC	AD
6	B7	B0	B1	[illegible]	B3	B4	B5
7	BF	B8	B9	BA	BB	BC	BD

RES b, (HL)
RES b, (IX + d)
RES b, (IY + d)

DDCB —
FDCB —

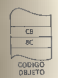
b:	0	1	2	3	4	5	6	7
	86	8E	96	9E	A6	AE	B6	BE

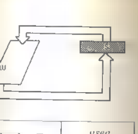
Banderas:

S	Z		H		P/V	N	C

(efecto nulo)

Ejemplo:





Estados T	μseg @ 2 MHz
8	4
15	7.5
23	11.5
23	11.5

d), (IY + d): indexado.

L
85
8D
95
9D
A5
AD
B5
BD

7
BE

o)

Ejemplo: RES 1, H

CB
8C
CODIGO OBJETO

Antes: H 42

Después: H 40

RET

Retorno de subrutina.

Función: $PC_{inf} \leftarrow (SP)$; $PC_{sup} \leftarrow (SP + 1)$; $SP \leftarrow SP + 2$

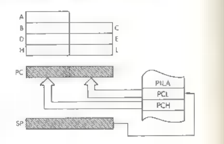
Formato:

1	1	0	0	1	0	0	1

C9

Descripción: El contador del programa se extrae de la pila tal como se describe en las instrucciones POP. La siguiente instrucción se toma de la posición señalada por el PC.

Flujo de datos:



Tiempo: 3 ciclos M; 10 estados T; 5 μseg @ 2 MHz.

Direccionamiento: Indirecto.

Banderas:

S	Z		H		P/V	N	C

(efecto nulo)

Ejemplo: RET

Antes:

PC 08B1

SP 3310

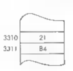
Después:

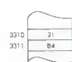
PC B421

SP 3312

C9

CODIGO OBJETO





RET cc

Función:

Formato:

Descripción:

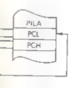
Flujo de datos:

Tiempo:

Direccionamiento:

Códigos byte:

Banderas:





RET cc

Retorno condicional de subrutina.

Función:

Si cc es verdadero: $PC_{inf} \leftarrow (SP); PC_{sup} \leftarrow (SP + 1); SP \leftarrow SP + 2$

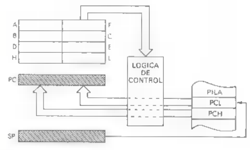
Formato:

1	1	←	cc	→	0	0	0

Descripción:

Si se satisface la condición, el contenido del contador del programa se extrae de la pila tal como se describe en las instrucciones POP. La siguiente instrucción se toma de la posición contenida en el PC. Si la condición no se satisface, la ejecución de instrucciones continúa en secuencia.

Flujo de datos:



cc puede ser:

NZ – 000	PO – 100
Z – 001	PE – 101
NC – 010	P – 110
C – 011	M – 111

Tiempo:

Condición satisfecha: 3 ciclos M; 11 estados T; 6.5 μseg @ 2 MHz.
Condición no satisfecha: 1 ciclo M; 5 estados T; 2.5 μseg @ 2 MHz.

Direccionamiento:

Indirecto.

Códigos byte:

CC:	NZ	Z	NC	C	PO	PE	P	M
	C0	C8	D0	D8	E0	E8	F0	F8

Banderas:



(efecto nulo)

Ejemplo: RET NC



Antes:

F: 00

PC: 0124

SP: 8511

8511	85
8512	B1

D0

CODIGO OBJETO

Después:

F: 00

PC: B185

SP: 8513

8511	85
8512	B1

RETI

Función:

Formato:

Descripción:

Flujo de datos:

Tiempo:

Direccionamiento:

Banderas:

RETI

Retorno de interrupción.

Función:

$PC_{inf} \leftarrow (SP)$; $PC_{sup} \leftarrow (SP + 1)$; $SP \leftarrow SP + 2$

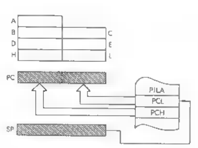
Formato:

1	1	1	0	1	1	0	1	byte 1: ED
0	1	0	0	1	1	0	1	byte 2: 4D

Descripción:

El contador del programa se extrae de la pila tal como se describe en las instrucciones POP. Los dispositivos periféricos Zilog reconocen esta instrucción como el final de una rutina de servicio a periférico para controlar adecuadamente las prioridades de interrupción internas. Para volver a habilitar las interrupciones es preciso ejecutar una instrucción EI antes de RETI.

Flujo de datos:



Tiempo:

4 ciclos M; 14 estados T; 7 μseg @ 2 MHz.

Direccionamiento:

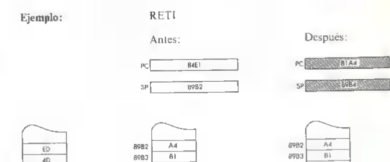
Indirecto.

Banderas:

S	Z		H		P/V	N	C

(efecto nulo)

Ejemplo: RETI



RETN

Función:

Formato:

Descripción:

Flujo de datos:

Tiempo:

Direccionamiento:

Banderas:

RETN

Retorno de una interrupción no enmascarable.

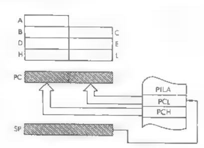
Función: $PC_{inf} \leftarrow (SP)$; $PC_{sup} \leftarrow (SP + 1)$; $SP \leftarrow SP + 2$; $IFF1 \leftarrow IFF2$

Formato:

1	1	1	0	1	1	0	1	byte 1: ED
0	1	0	0	0	1	0	1	byte 2: 45

Descripción: El contador del programa se extrae de la pila tal como se describe en las instrucciones POP. A continuación se copia en IFF1 el contenido de IFF2 (biestable de almacenamiento), para restaurar el estado de la bandera de interrupciones antes de una interrupción no enmascarable.

Flujo de datos:



Tiempo: 4 ciclos M; 14 estados T; 7 μseg @ 2 MHz.

Direccionamiento: Indirecto.

Banderas:

S	Z		H		P/V	N	C

(efecto nulo)

Ejemplo: RETN

Antes:

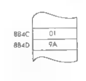
Después:

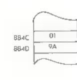












RL s

Función:

Formato: s: r

(HL)

(IX + d)

(IY + d)

Descripción:

Flujo de datos:

RL s

Rotación a la izquierda del operando s a través del acarreo.

Función:

C	←	7 ← 0	←
C		s	

Formato:

s:		
r	1 1 0 0 1 0 1 1	byte 1: CB
	0 0 0 1 0 ← r →	byte 2
(HL)	1 1 0 0 1 0 1 1	byte 1: CB
	0 0 0 1 0 1 1 0	byte 2: 16
(IX + d)	1 1 0 1 1 1 0 1	byte 1: DD
	1 1 0 0 1 0 1 1	byte 2: CB
	← d →	byte 3: valor del desplazamiento
	0 0 0 1 0 1 1 0	byte 4: 16
(IY + d)	1 1 1 1 1 1 0 1	byte 1: FD
	1 1 0 0 1 0 1 1	byte 2: CB
	← d →	byte 3: valor del desplazamiento
	0 0 0 1 0 1 1 0	byte 4: 16

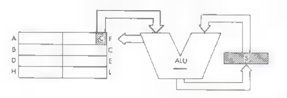
r puede ser:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Descripción:

El contenido de la posición del operando especificado se desplaza un bit hacia la izquierda. El contenido de la bandera de acarreo se lleva al bit 0, y el del bit 7 a la mencionada bandera. El resultado final vuelve a almacenarse en la posición de partida; s se define en la descripción de instrucciones RLC similares.

Flujo de datos:



Tiempo:

s	*Ciclos M*	*Estados T*	*μseg @ 2 MHz*
r	2	8	4
(HL)	4	15	7.5
(IX + d)	6	23	11.5
(IY + d)	6	23	11.5

Direccionamiento: r: implícito; (HL): indirecto; (IX + d), (IY + d): indexado.

Códigos byte: RL r

r:	A	B	C	D	E	H	L
CB	17	10	11	12	13	14	15

Banderas:

S	Z		H		P/V	N	C
●	●		0		●	0	●

C queda determinado por el bit 7 de la fuente.

Ejemplo: RL E



Antes:



Después:



RLA

Función:

Formato:

Descripción:

Flujo de datos:

Tiempo:

Direccionamiento:

Banderas:

Ejemplo:



RLA

Rotación a la izquierda del acumulador a través de la bandera de acarreo.

Función:



Formato:

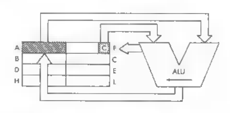
0	0	0	1	0	1	1	1

17

Descripción:

El contenido del acumulador se desplaza un bit hacia la izquierda. El contenido de la bandera de acarreo pasa al bit 0, y el del bit 7 a la mencionada bandera (rotación de 9 bits).

Flujo de datos:



Tiempo: 1 ciclo M; 4 estados T; 2 μseg @ 2 MHz.

Direccionamiento: Implícito.

Banderas:



C queda determinado por el bit 7 de A.

Ejemplo:

RLA

Antes:



Después:





RLCA

Rotación a la izquierda del acumulador con copia al acarreo

Función:



Formato:

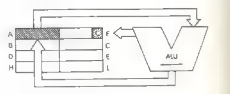


07

Descripción:

El contenido del acumulador se rota un bit hacia la izquierda. El contenido original del bit 7 se lleva a la bandera de acarreo y al bit 0.

Flujo de datos:



Tiempo: 1 ciclo M; 4 estados T; 2 μseg @ 2 MHz.

Direccionamiento: Implícito.

Banderas:



C queda determinado por el bit 7 de A.

Ejemplo:

RLCA

Antes:

A | 6B | 01 | F

Después:

A | D6 | 00 | F

07

CODIGO OBJETO

Nota: Con excepción de las banderas, esta instrucción es idéntica a RLC A. Se ha incluido para garantizar la compatibilidad con el 8080.

RLC r

Función:

Formato:

Descripción:

Flujo de datos:

Tiempo:

Direccionamiento:

Códigos byte: r:

Banderas:

RLC r

Rotación a la izquierda del registro r con copia al acarreo.

Función:



Formato:

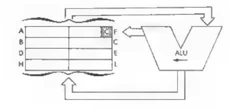
1	1	0	0	1	0	1	1	byte 1: CB
0	0	0	0	0	← r →			byte 2

Descripción:

El contenido del registro especificado se rota a la izquierda. El contenido original del bit 7 se lleva a la bandera de acarreo y al bit 0; r puede ser:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

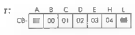
Flujo de datos:



Tiempo: 2 ciclos M; 8 estados T; 4 μseg @ 2 MHz.

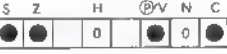
Direccionamiento: Implícito.

Códigos byte:



r:	A	B	C	D	E	H	L
CB-	[illegible]	00	01	02	03	04	[illegible]

Banderas:



S	Z		H		P/V	N	C
●	●		0		●	0	●

C queda determinado por el bit 7 del registro fuente.

Ejemplo: RLC B

	Antes:	Después:
B	62	C4
F	56	80

CB
00
CODIGO OBJETO

RLC (HL)

Rot…
copi…

Función:

Formato:

Descripción:

El …
co…
y e…
do…

Flujo de datos:

Tiempo: 4 …

Direccionamiento: In…

Banderas:

C…

Ejemplo: R…

A…

H

6114

CB
06
CODIGO OBJETO

RLC (HL)

Rotación a la izquierda de la posición de memoria (HL) con copia al acarreo.

Función:

C ◄— [7 ◄— 0] ◄— (HL)

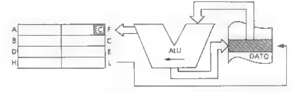
Formato:

1	1	0	0	1	0	1	1	byte 1: CB
0	0	0	0	0	1	1	0	byte 2: 06

Descripción:

El contenido de la posición de memoria direccionada por el contenido del par de registros (HL) se rota a la izquierda un bit, y el resultado vuelve a almacenarse en esa posición. El contenido del bit 7 se lleva a la bandera de acarreo y al bit 0.

Flujo de datos:



Tiempo: 4 ciclos M; 15 estados T; 7.5 μseg @ 2 MHz.

Direccionamiento: Indirecto.

Banderas:



C queda determinado por el bit 7 de la posición de memoria.

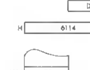
Ejemplo:

RLC (HL)

Antes:

D3 F

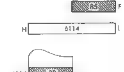
H 6114 L



Después:

85 F

H 6114 L





RLC (IX + d)

Rotación a la izquierda de la posición de memoria (IX + d) con copia al acarreo.

Función:

C ← 7 ← 0 (IX + d)

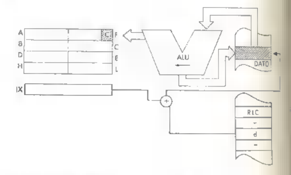
Formato:

1 1 0 1 1 1 0 1	byte 1: DD
1 1 0 0 1 0 1 1	byte 2: CB
← d →	byte 3: valor del desplazamiento
0 0 0 0 0 1 1 0	byte 4: 06

Descripción:

El contenido de la posición de memoria direccionada por el contenido del registro IX más el valor del desplazamiento dado se rota a la izquierda, y el resultado vuelve a almacenarse en la misma posición. El contenido del bit 7 se lleva a la bandera de acarreo y al bit 0.

Flujo de datos:



Tiempo: 6 ciclos M; 23 estados T; 11.5 μseg @ 2 MHz.

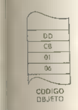
Direccionamiento: Indexado.

Banderas:



C queda determinado por el bit 7 de la posición de memoria

Ejemplo:



ción de memoria (IX + d) con

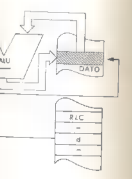
DD

CB

valor del desplazamiento

06

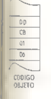
nemoria direccionada por el
or del desplazamiento dado se
elve a almacenarse en la mis-
se lleva a la bandera de aca-



g @ 2 MHz.

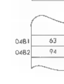
e la posición de memoria.

Ejemplo: RLC (IX + 1)



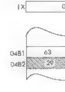
Antes:





Después:

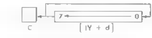
01 F



RLC (IY + d)

Rotación a la izquierda de la posición de memoria (IY + d) se copia al acarreo.

Función:



Formato:

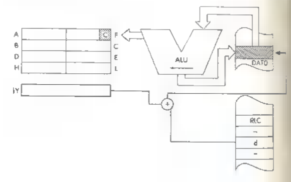


0 0 0 0 0 1 1 0 byte 4: 06

Descripción:

El contenido de la posición de memoria direccionada por el contenido del registro IY más el valor del desplazamiento dado se rota a la izquierda, y el resultado vuelve a almacenarse en la misma posición. El contenido del bit 7 se lleva a la bandera de acarreo y al bit 0.

Flujo de datos:



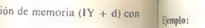
Tiempo: 6 ciclos M; 23 estados T; 11.5 μseg @ 2 MHz.

Direccionamiento: Indexado.

Banderas:



C queda determinado por el bit 7 de la posición de memoria.



FD

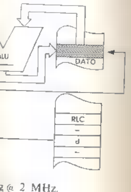
CB

valor del desplazamiento

06



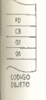
emoria direccionada por el
r del desplazamiento dado se
elve a almacenarse en la mis-
se lleva a la bandera de aca-



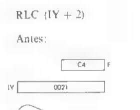
g @ 2 MHz.

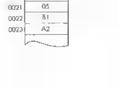
e la posición de memoria.

Ejemplo: RLC (IY + 2)

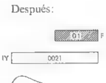


Antes:





Después:

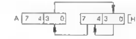




RLD

Rotación decimal a la izquierda.

Función:



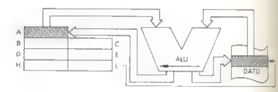
Formato:

1	1	1	0	1	1	0	1	byte 1: ED
0	1	1	0	1	1	1	1	byte 2: 6F

Descripción:

Los 4 bits inferiores de la posición de memoria direccionada por el contenido de HL se trasladan a los bits superiores de la misma posición. Los 4 bits superiores pasan a ocupar el lugar de los 4 inferiores del acumulador. A su vez, éstos pasan a ocupar el lugar de los 4 bits inferiores de la posición de memoria especificada originalmente. Todos estos movimientos se ejecutan simultáneamente.

Flujo de datos:



Tiempo: 5 ciclos M; 18 estados T; 9 µseg @ 2 MHz.

Direccionamiento: Indirecto.

Banderas:



Ejemplo:

RL

Ar

A

H

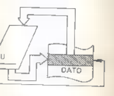


B4F2

ED

F

de memoria direccionada
a los bits superiores de la
es pasan a ocupar el lugar
A su vez, éstos pasan a
s de la posición de memo-
estos movimientos se eje-

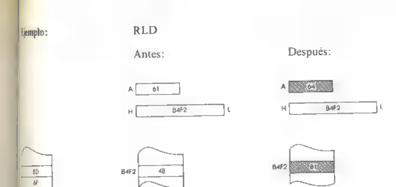


2 MHz.

Ejemplo:

RLD

Antes: Después:



RR s

Rotación a la derecha de s a través del acarreo.

Función:

[7 ——► 0] ► [C] (s, C)

Formato:

r	1	1	0	0	1	0	1	1	byte 1: CB
	0	0	0	1	1	←	r	→	byte 2
(HL)	1	1	0	0	1	0	1	1	byte 1: CB
	0	0	0	1	1	1	1	0	byte 2: 1E
(IX + d)	1	1	0	1	1	1	0	1	byte 1: DD
	1	1	0	0	1	0	1	1	byte 2: CB
	←			d				→	byte 3: valor del desplazamiento
	0	0	0	1	1	1	1	0	byte 4: 1E
(IY + d)	1	1	1	1	1	1	0	1	byte 1: FD
	1	1	0	0	1	0	1	1	byte 2: CB
	←			d				→	byte 3: valor del desplazamiento
	0	0	0	1	1	1	1	0	byte 4: 1E

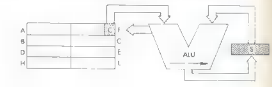
r puede ser:

A — 111	E — 011
B — 000	H — 100
C — 001	L — 101
D — 010	

Descripción:

El contenido de la posición determinada por el operando especificado se desplaza a la derecha. El contenido de la bandera de acarreo pasa al bit 7, y el del bit 0 a la mencionada bandera. El resultado final vuelve a almacenarse en la posición de partida; s se define en la descripción de instrucciones RLC similares.

Flujo de datos:



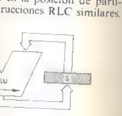
vés del acarreo.

CB

CB
E
DD
CB
alor del desplazamiento
E
D
B
alor del desplazamiento
E

– 011
– 100
– 101

da por el operando espe-
contenido de la bandera
a la mencionada bandera.
e en la posición de parti-
rucciones RLC similares.



Tiempo:

s	Ciclos M	Estados T	µseg @ 2 MHz
r	2	8	4
(HL)	4	15	7.5
(IX + d)	6	23	11.5
(IY + d)	6	23	11.5

Direccionamiento: r: implícito; (HL): indirecto; (IX + d), (IY + d): indexado.

Códigos byte: RR r:

r	A	B	C	D	E	H	L
CB	1F	18	19	1A	1B	1C	1D

Banderas:

S	Z		H		P/V	N	C
●	●		0		●	0	●

C queda determinado por el bit 0 del dato fuente.

Ejemplo: RR H

CB
1C
CODIGO OBJETO

Antes:

H 6B   41 L

Después:



RRA

Rotación a la derecha del acumulador a través del acarreo

Función:

Formato:

0	0	0	1	1	1	1	1

1F

Descripción:

El contenido del acumulador se desplaza un bit hacia la derecha. El contenido de la bandera de acarreo pasa al bit 7, y el del bit 0 a la mencionada bandera (rotación de 9 bits).

Flujo de datos:

Tiempo: 1 ciclo M; 4 estados T: 2 μseg a 2 MHz.

Direccionamiento: Implícito.

Banderas:

S	Z		H		P/V	N	C
			0			0	●

C queda determinado por el bit 0 de A.

Ejemplo:

RRA

Antes:

A	F4	95	F

Después:

A	FA	84	F

Nota: Esta instrucción es casi idéntica a RR A. Se incluye para garantizar la compatibilidad con el 8080.

RRC s

Ro

Función:

Formato:

s

r

(HL)

(IX + d)

(IY + d)

Descripción:

Flujo de datos:

RRC s

Rotación a la derecha de s con copia al acarreo.

Función:

Formato:

s puede ser: r, (HL), (IX + d), (IY + d).

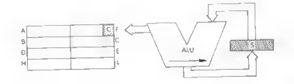
r	1	1	0	0	1	0	1	1	byte 1: CB
	0	0	0	0	1	←	r	→	byte 2
(HL)	1	1	0	0	1	0	1	1	byte 1: CB
	0	0	0	0	1	1	1	0	byte 2: 0E
(IX + d)	1	1	0	1	1	1	0	1	byte 1: DD
	1	1	0	0	1	0	1	1	byte 2: CB
	←			d				→	byte 3: valor del desplazamiento
	0	0	0	0	1	1	1	0	byte 4: 0E
(IY + d)	1	1	1	1	1	1	0	1	byte 1: FD
	1	1	0	0	1	0	1	1	byte 2: CB
	←			d				→	byte 3: valor del desplazamiento
	0	0	0	0	1	1	1	0	byte 4: 0E

r puede ser:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Descripción: El contenido de la posición determinada por el operando especificado se rota a la derecha, y el resultado vuelve a almacenarse en la misma posición. El contenido del bit 0 pasa a la bandera de acarreo y al bit 7; s se define en la descripción de instrucciones RLC similares.

Flujo de datos:



Tiempo:

s	*Ciclos M*	*Estados T*	*μseg @ 2 MHz*
r	2	8	4
(HL)	4	15	7.5
(IX + d)	6	23	11.5
(IY + d)	6	23	11.5

Direccionamiento: r: implícito; (HL): indirecto; (IX + d), (IY + d): indexado.

Códigos byte: RRC r

r	A	B	C	D	E	H	L
CB	0F	08	09	0A	0B	0C	0D

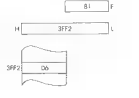
Banderas:

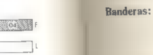


C queda determinado por el bit 0 del dato fuente.

Ejemplo: RRC (HL)

Antes: Después:







CB
0E
CODIGO OBJETO

RRCA

Ro

Función:

Formato:

Descripción:

E
c

Flujo de datos:

Tiempo:

Direccionamiento:

Banderas:

Ejemplo:



RRCA

Rotación a la derecha del acumulador con copia al acarreo.

Función:

7 ——→ 0 → C (A)

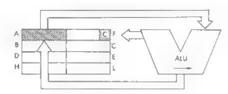
Formato:

0	0	0	0	1	1	1	1

0F

Descripción: El contenido del acumulador se rota un bit hacia la derecha. El contenido del bit 0 pasa a la bandera de acarreo y al bit 7.

Flujo de datos:



Tiempo: 1 ciclo M; 4 estados T; 2 μseg @ 2 MHz.

Direccionamiento: Implícito.

Banderas:

S	Z		H		P/V	N	C
			0			0	●

C queda determinado por el bit 0 de A.

Ejemplo: RRCA



Antes:



Después:



RRD

Rotación decimal a la derecha.

Función:



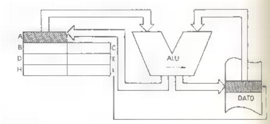
Formato:

1	1	1	0	1	1	0	1	byte 1: ED
0	1	1	0	0	1	1	1	byte 2: 67

Descripción:

Los 4 bits superiores de la posición de memoria direccionada por el contenido del par de registro HL pasan a ocupar el lugar de los 4 inferiores de esa posición. Estos 4 bits inferiores pasan, a su vez, a ocupar el lugar de los 4 inferiores del acumulador. A su vez, estos 4 se trasladan al lugar de los 4 superiores de la posición de memoria especificada originalmente. Todos estos movimientos se ejecutan simultáneamente.

Flujo de datos:



Tiempo: 5 ciclos M; 18 estados T; 9 μseg @ 2 MHz.

Direccionamiento: Indirecto.

Banderas:

S	Z		H		P/V	N	C
●	●		0		●	0	

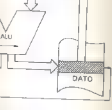
Ejemplo:



: ED

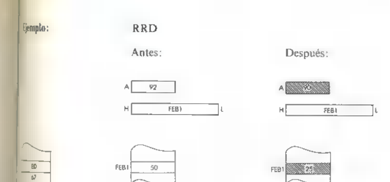
: 67

ón de memoria direccionada
o HL pasan a ocupar el lugar
Estos 4 bits inferiores pasan,
inferiores del acumulador. A
ar de los 4 superiores de la
originalmente. Todos estos
camente.



@ 2 MHz.

Ejemplo:



ED
67
CODIGO OBJETO

RST p

Reinicio en p.

Función:

$(SP-1) \leftarrow PC_{sup}$; $(SP-2) \leftarrow PC_{inf}$; $SP \leftarrow SP-2$; $PC_{sup} \leftarrow 0$; $PC_{inf} \leftarrow P$

Formato:

1	1	← p →	1	1	1

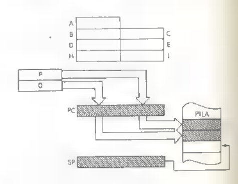
Descripción:

El contenido del contador del programa se empuja en la pila tal como se describe en las instrucciones PUSH. El valor especificado de p se carga en el PC, y la siguiente instrucción se toma de esta nueva dirección; p puede ser:

00H – 000	20H – 100
08H – 001	28H – 101
10H – 010	30H – 110
18H – 011	38H – 111

La instrucción ejecuta un salto a cualquiera de las ocho direcciones de inicio de la memoria inferior, y sólo necesita un byte. Puede utilizarse como respuesta rápida a una interrupción.

Flujo de datos:



Tiempo: 3 ciclos M; 11 estados T; 5.5 μseg @ 2 MHz.

Direccionamiento: Indirecto.

Códigos byte:

p:	00	08	10	18	20	28	30	38
	C7	CF	D7	DF	E7	EF	F7	FF

Banderas: S Z H P/V N C (efecto nulo)

Ejemplo: RST 38H

Antes:

PC 441A

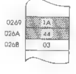
SP 026B

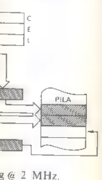
Después:

PC 0038

SP 0269







SBC A, s

Resta con acarreo del acumulador y el operando especificado

Función: A ← A − s − C

Formato: s puede ser: r, n, (HL), (IX + d) o (IY + d)

r	1 0 0 1 1 ←r→	
n	1 1 0 1 1 1 1 0	byte 1: DE
	←n→	byte 2: dato inmediato
(HL)	1 0 0 1 1 1 1 0	byte 1: 9E
(IX + d)	1 1 0 1 1 1 0 1	byte 1: DD
	1 0 0 1 1 1 1 0	9E
	←d→	byte 3: valor del desplazamiento
(IY + d)	1 1 1 1 1 1 0 1	byte 1: FD
	1 0 0 1 1 1 1 0	byte 2: 9E
	←d→	byte 3: valor del desplazamiento

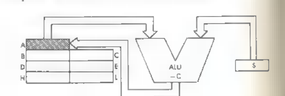
r puede ser:

A − 111	E − 011
B − 000	H − 100
C − 001	L − 101
D − 010	

Descripción: El operando especificado s, al que se suma el contenido de la bandera de acarreo, se resta del contenido del acumulador y el resultado se almacena en el acumulador; s se define en la descripción de instrucciones ADD similares.



Flujo de datos:



lor y el operando especificado.

) o (IY + d)

: DE

: dato inmediato

: 9E

: DD

valor del desplazamiento

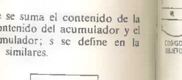
FD

9E

valor del desplazamiento

E – 011
H – 100
L – 101

se suma el contenido de la
ontenido del acumulador y el
mulador; s se define en la
similares.



Tiempo:

s	*Ciclos M*	*Estados T*	*μseg @ 2 MHz*
r	1	4	2
n	2	7	3.5
(HL)	2	7	3.5
(IX + d)	5	19	9.5
(IY + d)	5	19	9.5

Direccionamiento: r: implícito; n: inmediato; (HL): indirecto; (IX + d), (IY + d): indexado.

Códigos byte: SBC A, r

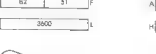
r: A	B	C	D	E	H	L
9F	98	99	9A	9B	9C	9D

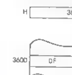
Banderas:

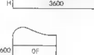
S	Z		H		P/V	N	C
●	●		●		●	1	●

Ejemplo: SBC A,(HL)

Antes: Después:







9E

CODIGO OBJETO

SBC HL, ss

Resta con acarreo de HL y el par de registros ss.

Función: HL ← HL − ss − C

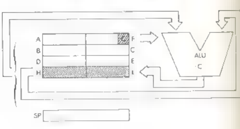
Formato:

1	1	1	0	1	1	0	1	byte 1: ED
0	1	s	s	0	0	1	0	byte 2

Descripción: El contenido del par de registros especificado, al que [illegible] suma e[illegible] de la bandera de acarreo, se resta del contenido del par de registros HL, y el resultado se almacena de nuevo [illegible] HL; [illegible] puede ser:

BC − 00   HL − 10
DE − 01   SP − 11

Flujo de datos:



Tiempo: 4 ciclos M; 15 estados T; 7.5 μseg @ 2 MHz.

Direccionamiento: Implícito.

Códigos byte:

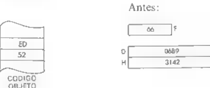
ss:	BC	DE	HL	SP
ED	42	52	62	72

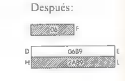
Banderas:

S	Z		H		P/V	N	C
●	●		?		●	1	●

H se activa [illegible] 1 si hay acarreo del bit 12.
C se activa a 1 si hay acarreo.

Ejemplo: SBC HL, DE





SCF

Po

Función: C

Formato:

Descripción: S

Tiempo: 1

Direccionamiento: I

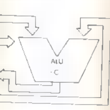
Banderas:

par de registros ss.

ED

especificado, al que se suma el
sta del contenido del par de
lmacena de nuevo en HL: ■

HL − 10
SP − 11



eg @ 2 MHz.

l bit 12.

pués:



SCF

Pone a 1 la bandera de acarreo.

Función: $C \leftarrow 1$

Formato:

0	0	1	1	0	1	1	1

37

Descripción: Se pone a 1 la bandera de acarreo.

Tiempo: 1 ciclo M; 4 estados T; 2 μseg @ 2 MHz.

Direccionamiento: Implícito.

Banderas:

S	Z		H		P/V	N	C
			0			0	1

SET b, s

Activa a 1 el bit b del operando s.

Función: $s_b \leftarrow 1$

Formato:

s:

r	1 1 0 0 1 0 1 1	byte 1: CB
	1 1 ← b → ← r →	byte 2
(HL)	1 1 0 0 1 0 1 1	byte 1: CB
	1 1 ← b → 1 1 0	byte 2
(IX + d)	1 1 0 1 1 1 0 1	byte 1: DD
	1 1 0 0 1 0 1 1	byte 2: CB
	← d →	byte 3: valor del desplazamiento
	1 1 ← b → 1 1 0	byte 4
(IY + d)	1 1 1 1 1 1 0 1	byte 1: FD
	1 1 0 0 1 0 1 1	byte 2: CB
	← d →	byte 3: valor del desplazamiento
	1 1 ← b → 1 1 0	byte 4:

r puede ser:

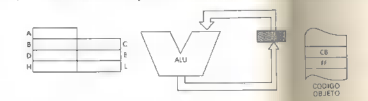
A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

b puede ser:

0 – 000	4 – 100
1 – 001	5 – 101
2 – 010	6 – 110
3 – 011	7 – 111

Descripción: El bit especificado de la posición determinada por s se activa a 1; s se define en la descripción de instrucciones BIT similares.

Flujo de datos:



Tiempo:

Direccionamiento:

Códigos byte:

Banderas:

Ejemplo:

do s.

1: CB

2

1: CB

2

1: DD

2: CB

3: valor del desplazamiento

4

: FD

: CB

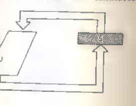
: valor del desplazamiento

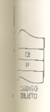
:

E – 011
H – 100
L – 101

4 – 100
5 – 101
6 – 110
7 – 111

determinada por s se activa a
e instrucciones BIT similares.





Tiempo:

s	Ciclos M	Estados T	μseg @ 2 MHz
r	2	8	4
(HL)	4	15	7.5
(IX + d)	6	23	11.5
(IY + d)	6	23	11.5

Direccionamiento: r: implícito; (HL): indirecto; (IX + d), (IY + d): indexado.

Códigos byte:

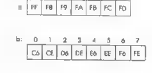
SET b, r

b: \ r:	A	B	C	D	E	H	L
0	C7	C0	C1	C2	C3	C4	C5
1	CF	C8	C9	CA	CB	CC	CD
2	D7	D0	D1	D2	D3	D4	D5
3	DF	D8	D9	DA	DB	DC	DD
4	E7	E0	E1	E2	E3	E4	E5
5	EF	E8	E9	EA	EB	EC	ED
6	F7	F0	F1	F2	F3	F4	F5
7	FF	F8	F9	FA	FB	FC	FD

SET b, (HL)

SET b, (IX + d)

SET b, (IY + d)



b:	0	1	2	3	4	5	6	7
	C6	CE	D6	DE	E6	EE	F6	FE

Banderas:

S	Z		H		P/V	N	C

(efecto nulo)

Ejemplo:

SET 7, A

Antes: A 61

Después: A E1

SLA s

Desplazamiento aritmético a la izquierda del operando s

Función:

C ← 7 ← 0 ← 0
(C = acarreo, s = operando)

Formato:

s:

r	1 1 0 0 1 0 1 1	byte 1: CB
	0 0 1 0 0 ← r →	byte 2
(HL)	1 1 0 0 1 0 1 1	byte 1: CB
	0 0 1 0 0 1 1 0	byte 2: 26
(IX + d)	1 1 0 1 1 1 0 1	byte 1: DD
	1 1 0 0 1 0 1 1	byte 2: CB
	← d →	byte 3: valor del desplazamiento
	0 0 1 0 0 1 1 0	byte 4: 26
(IY + d)	1 1 1 1 1 1 0 1	byte 1: FD
	1 1 0 0 1 0 1 1	byte 2: CB
	← d →	byte 3: valor del desplazamiento
	0 0 1 0 0 1 1 0	byte 4: 26

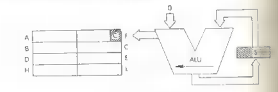
r puede ser:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Descripción:

El contenido de la posición de memoria determinada por el operando especificado se desplaza aritméticamente a la izquierda, pasando el contenido del bit 7 a la bandera de acarreo e introduciendo un 0 en el bit 0. El resultado final vuelve a almacenarse en la posición original; s se define en la descripción de instrucciones RLC similares.

Flujo de datos:



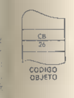
Tiempo:

Direccionamiento:

Códigos byte:

Banderas:

Ejemplo:



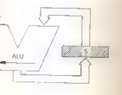
izquierda del operando s.

: CB

: CB
: 26
: DD
: CB
valor del desplazamiento
26
FD
CB
valor del desplazamiento
26

E – 011
H – 100
L – 101

memoria determinada por el
aritméticamente a la izquier-
a la bandera de acarreo e
El resultado final vuelve a
s se define en la descripción



Tiempo:

s	Ciclos M	Estados T	μseg @ 2 MHz
r	2	8	4
(HL)	4	15	7.5
(IX + d)	6	23	11.5
(IY + d)	6	23	11.5

Direccionamiento: r: implícito; (HL): indirecto; (IX + d), (IY + d): indexado.

Códigos byte: SLA r

r	A	B	C	D	E	H	L
CB	27	20	21	22	23	24	25

Banderas:

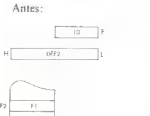
S	Z		H		P/V	N	C
●	●		0		●	0	●

C queda determinado por el bit 7 del dato fuente.

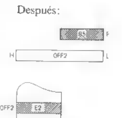
Ejemplo: SLA (HL)



Antes:



Después:



SRA s

Desplazamiento aritmético a la derecha del operando s.

Función:

7 → 0 → C
s

Formato:

s:

r	1 1 0 0 1 0 1 1	byte 1: CB
	0 0 1 0 1 ← r →	byte 2
(HL)	1 1 0 0 1 0 1 1	byte 1: CB
	0 0 1 0 1 1 1 0	byte 2: 2E
(IX + d)	1 1 0 1 1 1 0 1	byte 1: DD
	1 1 0 0 1 0 1 1	byte 2: CB
	← d →	byte 3: valor del desplazamiento
	0 0 1 0 1 1 1 0	byte 4: 2E
(IY + d)	1 1 1 1 1 1 0 1	byte 1: FD
	1 1 0 0 1 0 1 1	byte 2: CB
	← d →	byte 3: valor del desplazamiento
	0 0 1 0 1 1 1 0	byte 4: 2E

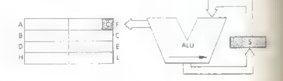
r puede ser:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Descripción:

El contenido de la posición determinada por el operando especificado se desplaza aritméticamente a la derecha. El contenido del bit 0 se traslada a la bandera de acarreo y el del bit 7 permanece invariable. El resultado final se almacena en la posición original; s se define en la descripción de instrucciones RLC similares.

Flujo de datos:



Tiempo:

Direccionamiento:

Códigos byte:

Banderas:

Ejemplo:

CB
2F
CODIGO OBJETO

derecha del operando s.

CB

CB

2E

DD

CB

valor del desplazamiento

2E

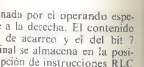
FD

CB

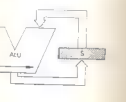
valor del desplazamiento

2E

E – 011
H – 100
L – 101



nada por el operando espe-
a la derecha. El contenido
de acarreo y el del bit 7
inal se almacena en la posi-
pción de instrucciones RLC



Tiempo:

s	Ciclos M	Estados T	μseg @ 2 MHz
r	2	8	4
(HL)	4	15	7.5
(IX + d)	6	23	11.5
(IY + d)	6	23	11.5

Direccionamiento: r: implícito; (HL): indirecto; (IX + d), (IY + d): indexado.

Códigos byte: SRA r



Banderas:



C queda determinado por el bit 0 del dato fuente.

Ejemplo: SRA A



Antes:

A | BB | D4 | F

Después:



SRL s

Desplazamiento lógico a la derecha de s.

Función:

0 → [7 → 0] → [ ]
s C

Formato:

s:

r	1	1	0	0	1	0	1	1	byte 1: CB
	0	0	1	1	1	← r →			byte 2
(HL)	1	1	0	0	1	0	1	1	byte 1: CB
	0	0	1	1	1	1	1	0	byte 2: 3E
(IX + d)	1	1	0	1	1	1	0	1	byte 1: DD
	1	1	0	0	1	0	1	1	byte 2: CB
	← d →								byte 3: valor del desplazamiento
	0	0	1	1	1	1	1	0	byte 4: 3E
(IY + d)	1	1	1	1	1	1	0	1	byte 1: FD
	1	1	0	0	1	0	1	1	byte 2: CB
	← d →								byte 3: valor del desplazamiento
	0	0	1	1	1	1	1	0	byte 4: 3E

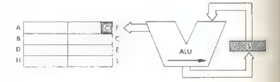
r puede ser:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Descripción:

El contenido de la posición determinada por el operando especificado se desplaza lógicamente a la derecha. Se introduce un cero en el bit 7, y el contenido del bit 0 pasa a la bandera de acarreo. El resultado final se almacena en la posición original.

Flujo de datos:



Tiempo:

Direccionamiento:

Códigos byte:

Banderas:

Ejemplo:

CB
3B
CODIGO OBJETO

cha de s.

CB

CB

3E

DD

CB

valor del desplazamiento

3E

FD

CB

valor del desplazamiento

3E

E – 011
H – 100
L – 101

nada por el operando espe-
a derecha. Se introduce un
bit 0 pasa a la bandera de
na en la posición original.

ALU

S

Tiempo:

s	Ciclos M	Estados T	µseg @ 2 MHz
r	2	8	4
(HL)	4	15	7.5
(IX + d)	6	23	11.5
(IY + d)	6	23	11.5

Direccionamiento: r: implícito; (HL): indirecto; (IX + d), (IY + d): indexado.

Códigos byte: SRL r

r:	A	B	C	D	E	H	L
CB	3F	38	39	3A	3B	3C	3D

Banderas:

S	Z		H		P/V	N	C
●	●		0		●	0	●

C queda determinado por el bit 0 del dato fuente.

Ejemplo: SRL E

CB
3B
CODIGO OBJETO

Antes:

01 F

02 E

Después:





SUB s

Resta del operando s del acumulador.

Función: A ← A − s

Formato: s puede ser: r, n, (HL), (IX + d) o (IY + d)

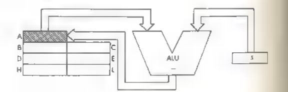
r	1 0 0 1 0 ←r→	
n	1 1 0 1 0 1 1 0	byte 1: D6
	←n→	byte 2: dato inmediato
(HL)	1 0 0 1 0 1 1 0	96
(IX + d)	1 1 0 1 1 1 0 1	byte 1: DD
	1 0 0 1 0 1 1 0	byte 2: 96
	←d→	byte 3: valor del desplazamiento
(IY + d)	1 1 1 1 1 1 0 1	byte 1: FD
	1 0 0 1 0 1 1 0	byte 2: 96
	←d→	byte 3: valor del desplazamiento

r puede ser:

A − 111	E − 011
B − 000	H − 100
C − 001	L − 101
D − 010	

Descripción: El operando especificado s se resta del acumulador, y el resultado se almacena en éste. El operando s se define en la descripción de instrucciones ADD similares.

Flujo de datos:





Tiempo:

Direccionamiento:

Códigos byte:

Banderas:

Ejemplo:



dor.

(IY + d)

06

lato inmediato

DD

6

alor del desplazamiento

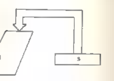
D

6

alor del desplazamiento

– 011
– 100
– 101

l acumulador. y el resulta-
se define en la descrip-
.



Tiempo:

s	*Ciclos M*	*Estados T*	*μseg @ 2 MHz*
r	1	4	2
n	2	7	3.5
(HL)	2	7	3.5
(IX + d)	5	19	9.5
(IX + d)	5	19	9.5

Direccionamiento: r: implícito; n: inmediato; (HL): indirecto; (IX + d), (IY + d): indexado.

Códigos byte: SUB r

r:	A	B	C	D	E	H	L
	97	90	91	92	93	94	95

Banderas:

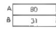
S	Z		H		P/V	N	C
●	●		●		●	1	●

Ejemplo: SUB B

90

CODIGO OBJETO

Antes:



Después:





XOR s

O exclusiva del acumulador y s.

Función: A ← A ⊕ s

Formato: s puede ser: r, n, (HL), (IX + d), o (IY + d)

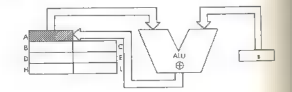
r	1 0 1 0 1 ← r →	
n	1 1 1 0 1 1 1 0	byte 1: EE
	← n →	byte 2: dato inmediato
(HL)	1 0 1 0 1 1 1 0	AE
(IX + d)	1 1 0 1 1 1 0 1	byte 1: DD
	1 0 1 0 1 1 1 0	byte 2: AE
	← d →	byte 3: valor del desplazamiento
(IY + d)	1 1 1 1 1 1 0 1	byte 1: FD
	1 0 1 0 1 1 1 0	byte 2: AE
	← d →	byte 3: valor del desplazamiento

r puede ser:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Descripción: El acumulador y el operando especificado s se someten a la operación "O" exclusiva, y el resultado se almacena en el acumulador; s se define en la descripción de instrucciones ADD similares.

Flujo de datos:



Tiempo:

Direccionamiento:

Códigos byte:

Banderas:

Ejemplo:



o (IY + d)

EE

dato inmediato

DD

AE

valor del desplazamiento

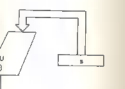
FD

AE

valor del desplazamiento

E – 011
H – 100
L – 101

cificado s se someten a la
ado se almacena en el acu-
ión de instrucciones ADD



Tiempo:

s	*Ciclos M*	*Estados T*	*μseg @ 2 MHz*
r	1	4	2
n	2	7	3.5
(HL)	2	7	3.5
(IX + d)	5	19	9.5
(IY + d)	5	19	9.5

Direccionamiento: r: implícito; n: inmediato; (HL): indirecto; (IX + d), (IY + d): indexado.

Códigos byte: XOR r



Banderas:



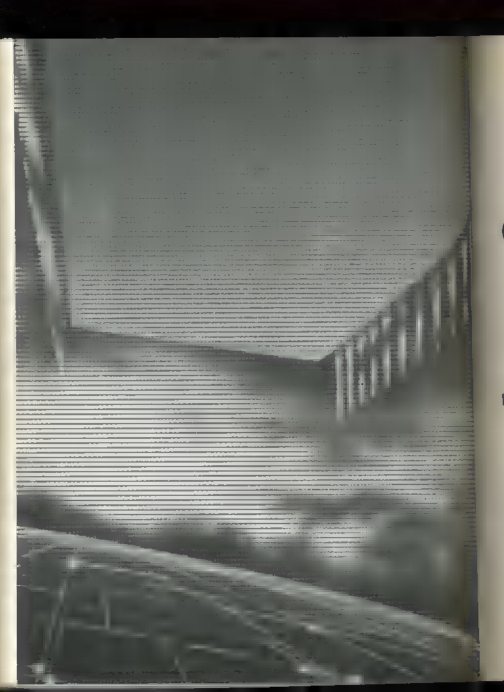
Ejemplo: XOR 81H

81
CODIGO OBJETO

Antes: A 36

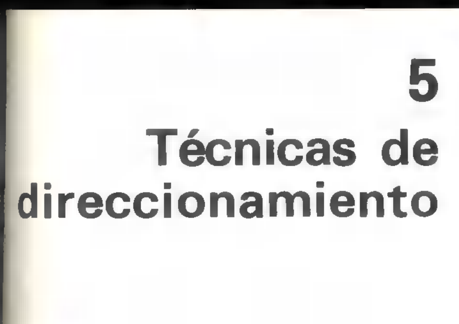
Después: A 87





direc

Introducción



5 Técnicas de direccionamiento

Introducción

Veremos en este capítulo la teoría general del direccionamiento y las diversas técnicas desarrolladas para facilitar la recuperación de datos. Pasaremos a continuación a examinar los mecanismos de direccionamiento específicos del Z80, centrándonos en particular en sus ventajas y sus inconvenientes. En la última parte dedicada a las aplicaciones se familiarizará al lector con los posibles intercambios entre diferentes técnicas de direccionamiento.

Como el Z80 tiene varios registros de 16 bits, además del contador del programa, que sirven para especificar direcciones, es importante que el usuario del mismo conozca los diversos modos de direccionamiento y, en particular, el uso del registro de índice. En una primera fase puede prescindirse de los procedimientos más complejos, pero hay que tener en cuenta que para desarrollar programas para este microprocesador son útiles todas las formas de direccionamiento. Pasemos ya a estudiar las opciones disponibles.

Modos de direccionamiento

Se llama *direccionamiento* a la especificación, dentro de una instrucción, de la posición del operando sobre el que actuará la misma. Los modos de direccionamiento que vamos a estudiar ahora se ilustran en la figura 5.1.

DIRECCIONAMIENTO IMPLICITO (O "POR REGISTRO")

Las instrucciones que operan exclusivamente con registros suelen utilizar el *direccionamiento implícito*, como ilustra la figura 5.1. El nombre se debe a que la instrucción no contiene de forma explícita la dirección del operando, sino que su código de operación especifica uno o más registros, por lo general el acumulador y algún otro u otros. Dado que no hay muchos registros internos (habitualmente ocho), esta técnica exige pocos bits; de hecho, bastan tres bits dentro de la instrucción para señalar a uno de los ocho registros internos; por tanto, tales instrucciones pueden, por lo general, codificarse totalmente con ocho bits. Esto constituye una ventaja importante, porque una instrucción de un byte se ejecuta casi siempre más rápidamente que otra de dos o tres.

Veamos un ejemplo de instrucción implícita:

LD A, B

que especifica "transferir el contenido de B a A" (carga de A a partir de B).

DIRECCIONAMIENTO INMEDIATO

También se muestra en la figura 5.1. Al código de operación de ocho bits sigue un literal (una constante) de 8 ó 16 bits. Este tipo de instrucción es necesario para, por ejemplo, cargar un valor de 8 bits en un registro de 8 bits. Como el microprocesador dispone de registros de 16 bits, será también necesario cargar literales de esa longitud. He aquí una instrucción inmediata:

ADD A, 0H

La segunda palabra de la instrucción contiene el literal "0", que se suma al acumulador.

Figura 5.1
Modos de direccionamien[to]
fundamentales.

specificación, dentro de una
ando sobre el que actuará la
iento que vamos a estudiar

TO (O "POR REGISTRO")

xclusivamente con registros
plícito, como ilustra la figu-
instrucción no contiene de
ando, sino que su código de
egistros, por lo general el
Dado que no hay muchos
o), esta técnica exige pocos
tro de la instrucción para
internos; por tanto, tales
codificarse totalmente con
ja importante, porque una
i siempre más rápidamente

ón implícita:

de B a A" (carga de A a

TO

.1. Al código de operación
tante) de 8 ó 16 bits. Este
, por ejemplo, cargar un
. Como el microprocesa-
, será también necesario
qui una instrucción inme-

ón contiene el literal "0",

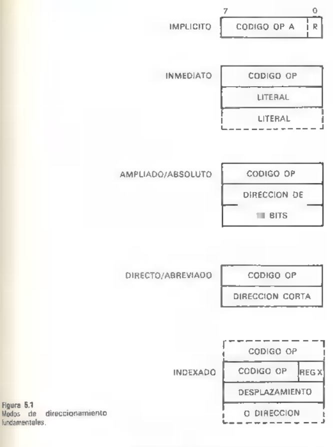


Figura 5.1
Modos de direccionamiento fundamentales.

DIRECCIONAMIENTO ABSOLUTO

Por lo general, hace referencia a la recuperación o al almacenamiento de datos de o en la memoria. Al código de operación sigue una dirección de 16 bits, de manera que esta forma de direccionamiento obliga a trabajar con instrucciones de tres bytes. Veamos un ejemplo:

LD (1234H), A

Especifica que debe almacenarse el contenido del acumulador en la posición hexadecimal de memoria "1234".

El inconveniente de esta forma de direccionamiento es la mencionada obligación de trabajar con instrucciones de tres bytes. Para mejorar la eficacia del microprocesador, se cuenta, a veces, con otro modo de direccionamiento llamado direccionamiento directo, que sólo necesita una palabra.

DIRECCIONAMIENTO DIRECTO (O "ABREVIADO")

En este modo, al código de operación sigue una dirección de 8 bits (también aparece en la figura 5.1). Tiene la ventaja de que equivale al direccionamiento absoluto, pero sólo con dos bytes en lugar de tres. El inconveniente es que está limitado a las direcciones comprendidas entre 0 y 255 o entre − 128 y + 127. En el primer caso ("página cero"), se habla también de direccionamiento abreviado o direccionamiento de página 0. Cuando se dispone de esta posibilidad, al direccionamiento absoluto se le llama, por contraste, *direccionamiento ampliado*. El intervalo − 128 a + 127 se utiliza en las instrucciones de bifurcación, y se llama direccionamiento relativo.

DIRECCIONAMIENTO RELATIVO

Las instrucciones de salto o bifurcación normal necesitan 8 bits para el código de operación más 16 bits para la dirección a la que debe saltar el programa. Como en el ejemplo anterior, este modo tiene el inconveniente de que obliga a usar tres palabras; por tanto, tres ciclos de memoria. El *direccionamiento relativo* se conforma con dos, y garantiza una bifurcación más rápida. La primera palabra es la especificación de la posición de salto, comprendida, por lo general, en la verificación que se efectúa; la segunda es un desplazamiento, positivo o negativo, que permite a la instrucción avanzar hasta 127 posiciones (siete bits) o retroceder hasta 128 (por lo general, de + 129 a − 126, porque el PC se habrá incrementado en 2). Como casi todos los bucles son breves, la mayor parte de ellos pueden controlarse por direccionamiento relativo, lo que mejora considerablemente la velocidad y la eficacia de tales subrutinas. Como ejemplo, ya hemos utilizado la instrucción JR NC, que especifica "saltar si no hay acarreo" a una posición situada a no más de 127 bytes de la instrucción de bifurcación (más exactamente, comprendida entre + 129 y − 126).

Figura 5.2
Direccionamiento (preindexación).

Esta técnica tiene dos ventajas: mayor velocidad, debida al uso de menos bytes, y facilidad de redireccionamiento del programa, que no depende de direcciones absolutas.

DIRECCIONAMIENTO INDEXADO

Se emplea este modo para acceder en secuencia a todos los elementos de un bloque o de una tabla; demostraremos la técnica hacia el final de este capítulo con ayuda de algunos ejemplos. La instrucción especifica una dirección y un registro de índice, cuyo contenido se suma a la dirección para obtener la dirección definitiva. De esta forma, la dirección puede ser el principio de una tabla situada en memoria, que se recorre en secuencia elemento tras elemento por medio del registro de índice (naturalmente, hacen falta instrucciones de incremento y decremento para dicho registro). En la práctica, el tamaño del registro de índice, el de la dirección y el del campo de desplazamiento suelen tener un límite.

PREINDEXACION Y POSTINDEXACION

La preindexación es la forma de indexación normal; en ella, la dirección final es la suma de un desplazamiento o dirección y del registro de índice. Se muestra en la figura 5.2, con un campo de desplazamiento de 8 bits y un registro de índice de 16 bits.

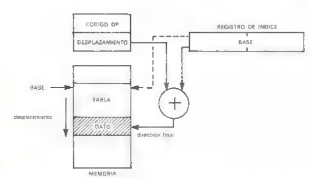


Figura 5.2
Direccionamiento (preindexación)

La postindexación considera el contenido del campo de desplazamiento como la *dirección* del desplazamiento real en lugar de como el desplazamiento propiamente dicho, como

ilustra la figura 5.3. En esta modalidad, la dirección final es la suma del contenido del registro de índice y de la palabra de memoria *designada por el campo de desplazamiento*. Esta técnica es, en realidad, una combinación de direccionamiento indirecto, que estudiaremos a continuación, y preindexación.

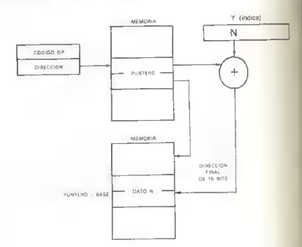


Figura 5.3
Direccionamiento indexado indirecto (postindexación).

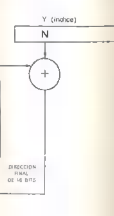
DIRECCIONAMIENTO INDIRECTO

Ya hemos visto que a veces dos subrutinas tienen que intercambiar un gran volumen de datos almacenados en memoria. Todavía es más frecuente que varios programas, o varias subrutinas, necesiten acceso a un bloque de información común. Para proteger el cuerpo del programa es aconsejable que dicho bloque no se mantenga en una posición fija de memoria; en efecto, el tamaño del mismo puede aumentar o disminuir a lo largo del tiempo, por lo que debe residir en diversas zonas de la memoria, según su volumen. En estas circunstancias no sería práctico acceder al bloque mediante direccionamiento absoluto, porque habría que escribir el programa completo cada vez que cambiase de posición.

La solución está en situar la dirección de partida del bloque en una posición fija de memoria. Es una solución similar a la adoptada cuando varias personas necesitan tener acceso a una casa de la que sólo hay una llave: esconder ésta debajo del

Figura 5.4
Direccionamiento indirecto.

ad, la dirección final es la índice y de la palabra de *esplazamiento*. Esta técnica direccionamiento indirecto, preindexación.



TO

ubrutinas tienen que inter- almacenados en memoria. rogramas, o varias subru- información común. Para consejable que dicho blo- ija de memoria; en efecto, o disminuir a lo largo del versas zonas de la memo- nstancias no sería práctico amiento absoluto, porque leto cada vez que cambia-

ión de partida del bloque una solución similar a la esitan tener acceso a una esconder ésta debajo del

felpudo. Todos los usuarios saben lo que tienen que hacer para dar con la llave (mirar bajo el felpudo) o con la dirección en la que se celebrará una reunión (este caso está más próximo al que nos ocupa). Por tanto, el direccionamiento indirecto utiliza habitualmente un código de operación al que sigue una dirección de 16 bits, que se utiliza para recuperar una palabra de la memoria. Será casi siempre una palabra de 16 bits (en nuestro caso, dos bytes), puesto que se trata de una dirección; la situación se ilustra en la figura 5.4: los dos bytes de la dirección especificada A1 contienen "A2", que se interpreta como la verdadera dirección del dato al que se desea acceder.

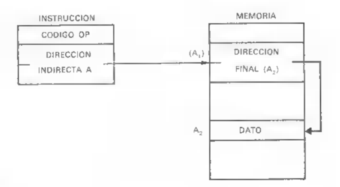


Figura 5.4
Direccionamiento indirecto.

El direccionamiento indirecto es particularmente útil en todas las situaciones en que se emplean punteros, porque así las diversas áreas del programa pueden dirigirse a dichos punteros para acceder a una palabra o a un bloque de datos con comodidad y elegancia. La dirección final también puede obtenerse señalando dentro de la instrucción un registro de 16 bits en el que esté contenida; esto se llama "registro indirecto".

COMBINACIONES DE MODOS

Las técnicas de direccionamiento que hemos visto pueden combinarse. En un esquema de direccionamiento completamente general debe ser posible trabajar en varios niveles; así, la dirección A2 podría interpretarse, a su vez, como una dirección indirecta, etc.

El direccionamiento indexado puede combinarse con el acceso indirecto, lo que permite acceder rápidamente a la palabra a de un bloque de datos si se sabe dónde está el puntero que señala la dirección de partida (véase figura 5.2).

Una vez familiarizados con los modos de direccionamiento habituales, estudiaremos los del Z80, que ofrece bastantes posibilidades. Hay que tener en cuenta que los microprocesadores, debido a la limitación de la complejidad del μP, que ha de estar contenida en una sola pastilla, suelen ofrecer tan sólo un pequeño subconjunto de las técnicas estudiadas.

Modos de direccionamiento del Z80

DIRECCIONAMIENTO IMPLICITO (Z80)

Sobre todo, se emplea en instrucciones de un byte que operan con registros internos. Estas instrucciones se ejecutan en un solo ciclo de máquina.

Utilizan direccionamiento implícito (o "por registro"): LD r, r'; ADD A, r; ADC A,s; SUB s; SBC A,s; AND s; OR s; XOR s; CP s; INC r.

Zilog diferencia entre "direccionamiento por registro" y "direccionamiento implícito". Según este enfoque, el direccionamiento implícito se limita a las instrucciones que no tienen un campo específico para señalar a un registro interno, lo que introduce, en realidad, un nuevo modo de direccionamiento. Estos, de hecho, no bastan para definir las posibilidades de un microprocesador.

DIRECCIONAMIENTO INMEDIATO (Z80)

Como el Z80 dispone de registros de longitud simple (8 bits) y de pares de registros de longitud doble (16 bits), ofrece dos tipos de direccionamiento inmediato con literales de 8 o de 16 bits e instrucciones de dos o tres bytes. El segundo byte (y a veces, el tercero) contiene el código de operación seguido de la constante o literal que ha de cargarse en un registro o utilizarse en una operación. Son excepciones LD IX y LD IY, que necesitan códigos de operación de 16 bits.

Son ejemplos de instrucciones que utilizan direccionamiento inmediato:

LD R, n (dos bytes).
LD dd,nn (tres bytes).

y

ADD A,n (dos bytes).

Cuando el literal tiene dos bytes, el modo se llama en el Z80 "inmediato ampliado".

DIRECCIONAMIENTO ABSOLUTO O "AMPLIADO" (Z80)

El direccionamiento absoluto precisa, por definición, tres bytes; el primero es el código de operación, y los dos siguientes, la dirección de 16 bits que especifica la posición de memoria (la "dirección absoluta").

Por contraste con el "direccionamiento directo" (dirección de 8 bits), este modo se llama también "direccionamiento ampliado".

Utilizan, entre otras, direccionamiento ampliado las instrucciones

LD HL,(nn) y JP nn

donde nn representa la dirección de memoria de 16 bits y (nn) el contenido de la posición especificada.

DIRECCIONAMIENTO DE PAGINA CERO MODIFICADO (Z80)

El Z80 sólo dispone de direccionamiento de página cero para la instrucción RST, y se llama «direccionamiento de página cero modificado".

Dicha instrucción contiene un campo de 3 bits en las posiciones 5, 4 y 3 que señala una de las ocho posiciones de la memoria de página 0. La dirección real es $b_5b_4b_3000$, y se carga en el PC. Como sólo ocupa un byte, la instrucción se ejecuta rápidamente, y se genera fácilmente en el soporte físico. Suele emplearse para responder a interrupciones múltiples (hasta ocho). Tiene el inconveniente de que, o bien se limita la secuen-

cia de ejecución a ocho posiciones, o bien se combina con un salto que elimina la ventaja de la velocidad, porque las ocho direcciones de bifurcación están separadas por una distancia de 8 bytes.

DIRECCIONAMIENTO RELATIVO (Z80)

Por definición, el direccionamiento relativo exige dos bytes: el primero es el código de operación de "salto relativo", el segundo especifica el desplazamiento y su signo.

Esta instrucción se codifica "JR" para diferenciarla de la de salto absoluto.

Desde el punto de vista del tiempo de ejecución, la instrucción debe considerarse con cuidado. Si el resultado de una verificación es negativo, es decir, si no hay salto, consume sólo siete ciclos T, porque el contador del programa está ya señalando la instrucción siguiente.

Pero si el resultado de la comprobación es afirmativo, es decir, si hay salto, la instrucción necesita 12 estados T, porque hay que calcular una nueva dirección y cargarla [illegible] el contador del programa.

Para calcular lo que tarda en ejecutarse [illegible] segmento de programa, hay que andarse con cuidado. Cuando no se tiene la seguridad de si un salto va o no a efectuarse, téngase en cuenta que la instrucción necesitará en unos casos *12 estados T (la condición se satisface)* y en otros sólo 7 *(la condición no se satisface)*.

Por tanto, al escribir un bucle se conseguirá mayor rapidez de ejecución con una instrucción JR (salto relativo) [illegible] la condición que ha de verificarse *no suele* cumplirse (por ejemplo, la condición de que el contador no sea 0).

Si JR's se emplea fuera de un bucle y la condición que se verifica es desconocida, la duración suele calcularse a partir de un tiempo medio.

Este problema del tiempo no ocurre en el salto incondicional JR e, que no verifica ninguna condición y dura siempre 12 estados T.

DIRECCIONAMIENTO INDEXADO (Z80)

Este modo de direccionamiento no existía en el 8080, y constituye una novedad del Z80 (al igual que los dos registros de índice). En consecuencia, es necesario añadir un byte extra al código de operación de este microprocesador, que pasa a tener

Figura 5.5
Direccionamiento indexado con código de operación de [illegible] bytes.

16 bits (LDIR es otro ejemplo de código de operación de 16 bits). En la figura 5.5 se muestra la estructura de una instrucción indexada.

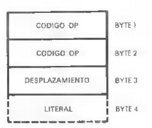


Figura 5.5
Direccionamiento indexado con código de operación de [illegible] bytes

Son instrucciones compatibles con el direccionamiento indexado

LD, ADD, INC, RLC, BIT, SET, CP y algunas otras.

Es una técnica que se utiliza mucho en programas que manipulan constantemente bloques de datos, tablas o listas.

DIRECCIONAMIENTO INDIRECTO (Z80)

El Z80 tiene cierta capacidad de direccionamiento indirecto, que se llama "direccionamiento indirecto por registro". En este modo, los pares de registros de 16 bits BC, DE y HL pueden emplearse como direcciones de memoria.

Cuando señalan un dato de 16 bits, lo hacen siempre a su mitad inferior; la superior reside en la dirección siguiente.

COMBINACIONES DE MODOS

Básicamente no existen, aunque las instrucciones que se refieren a dos operandos pueden emplear una forma de direccionamiento diferente para cada uno de ellos.

Así, una instrucción de carga o aritmética puede acceder a un operando en modo inmediato y a otro por medio de un índice.

Como veremos en el próximo apartado, el mecanismo de direccionamiento de bit puede acceder a un byte de 8 bits mediante tres modos de direccionamiento. Los mecanismos compatibles con las diferentes instrucciones se incluyeron en las descripciones del capítulo anterior.

DIRECCIONAMIENTO DE BIT

Dado que por direccionamiento se entiende el acceso a un byte, este mecanismo no se considera como tal, aunque, en cualquier caso, se trata de un recurso valioso. En la nomenclatura de Zilog sí que se describe como un "modo de direccionamiento", y aquí nos atendremos a ese punto de vista. Es una característica específica del Z80 que no existe en el 8080.

El direccionamiento de bit es un mecanismo de acceso a bits específicos. El Z80 dispone de instrucciones especiales para activar, desactivar y verificar bits específicos de un registro o una posición de memoria. El byte afectado admite el acceso por tres modos de direccionamiento: registro, registro indirecto e indexado. Dentro del código de operación se usan tres bits para seleccionar uno de los ocho bits.

Empleo de los modos de direccionamiento del Z80

DIRECCIONAMIENTO LARGO Y ABREVIADO

Ya hemos utilizado instrucciones de salto relativo en varios programas, instrucciones que se explican por sí mismas. Pero se plantea una pregunta interesante: ¿qué podemos hacer si el intervalo permisible de bifurcación no es suficiente para nuestras necesidades? En muchos microprocesadores la solución está en usar el llamado *salto largo*, que no es sino el salto a una posición de memoria que contiene una especificación de salto absoluta o "larga":

```
JR NC,$ + 3          BIFURCACION A LA DIRECCION
                     EN CURSO + 3, SI C ES CERO
JP LEJOS             SALTO A LEJOS, EN CASO CON-
                     TRARIO
(INSTRUCCION SIGUIENTE)
```

El anterior programa de dos líneas provoca la bifurcación a la posición LEJOS siempre que el acarreo no sea nulo. En el caso del Z80 puede usarse JP en lugar de JR para verificar todas las condiciones y solucionar este problema.

Figura 5.6
Diagrama de flujo de búsqueda de carácter.

USO DE LA INDEXACION PARA ACCEDER A BLOQUES SECUENCIALES

La indexación se usa, sobre todo, para direccionar las posiciones sucesivas de una tabla, con la sola limitación de contar con una longitud máxima inferior a 256 para que el desplazamiento pueda residir en un registro de índice de 8 bits.

Ya hemos aprendido a buscar un carácter. Ahora investigaremos una tabla de 100 elementos para averiguar si contiene el elemento "*". La dirección de partida de dicha tabla, que sólo tiene 100 elementos, se llama BASE. El programa aparece aquí debajo, y el diagrama de flujo correspondiente, en la figura 5.6.

```
BUSCAR     LD    IX, BASE
           LD    A, "*"
           LD    B, CUENTA
PRUEBA     CP    (IX)
           JR    Z, LOHALLE
           INC   IX
           DEC   B
           JR    NZ, PRUEBA
LOHALLE    ...
```

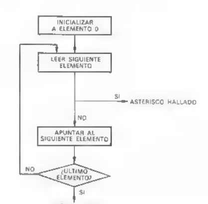


Figura 5.6
Diagrama de flujo de búsqueda de carácter.

En la sección de transferencia de bloques veremos una versión mejorada con la instrucción DJNZ.

RUTINA DE TRANSFERENCIA DE BLOQUES PARA MENOS DE 256 ELEMENTOS

Llamaremos CUENTA al número de elementos del bloque que deseamos transferir, número que se supone inferior a 256. DESDE es la dirección base del bloque. HASTA es la base del área de memoria a la que debe llevarse. El algoritmo es muy sencillo: se trata de mover las palabras una por una y de mantener un registro de la que estamos moviendo, almacenando su posición en el contador C. El programa es éste:

```
MOVBLQ  LD   IX, DESDE
        LD   IY, HASTA
        LD   C, CUENTA
BUCLE   LD   A, (IX)       TOMAR PALA-
                           BRA
        LD   (IY), A
        INC  IX
        INC  IY
        DEC  C
        JR   NZ, BUCLE
```

Empecemos a analizarlo por la primera parte:

```
MOVBLQ  LD   IX, DESDE
        LD   IY, HASTA
        LD   C, CUENTA
```

Estas tres instrucciones inicializan los registros IX, IY y C, respectivamente, como ilustra la figura 5.7. El registro de índice IX se usa como apuntador de fuente y se incrementa regularmente. El IY es el apuntador de destino, y también se incrementa con regularidad. En el registro C se carga el número de elementos que va a transferirse (limitado a un máximo de 256, porque se trata de un registro de 8 bits) y se decrementa regularmente. Cuando C alcance el valor 0, todos los elementos habrán sido transferidos. Las dos instrucciones que vienen a continuación

```
BUCLE   LD   A, (IX)
        LD   (IY), A
```

cargan el contenido de la posición de memoria señalada por IX en el acumulador, y a continuación lo transfieren a la posición de memoria indicada por el registro IY; en otras palabras,

Figura 5.7
Transferencia de bloques: inicialización del registro.

estas dos instrucciones transfieren un elemento del bloque fuente al bloque destino. A continuación se incrementan los dos registros de índice:

```
INC  IX
INC  IY
```

y se decrementa el registro contador:

```
DEC  C
```

Por último, si el contador no es 0, se vuelve a empezar en bucle:

```
JR  NZ, BUCLE
```

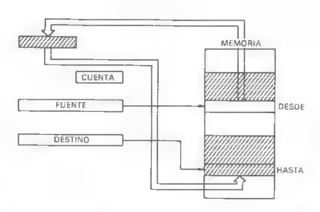


Figura 5.7
Transferencia de bloques: inicialización del registro.

El programa es un ejemplo de una posible aplicación de los registros de índice. Vamos a compararlo ahora con otro igual escrito para el microprocesador 6502 de tecnología MOS, que también dispone de capacidad de indexación, pero que sigue convenciones diferentes (es decir, que tiene límites de indexación diferentes). El programa en cuestión es el siguiente:

```
        LDX   # NUMERO
BUCLE   LDA   DESDE, X
        STA   HASTA, X
        DEX
        BNE   BUCLE
```

Sin necesidad de entrar en detalles, salta a la vista que es mucho más corto que el anterior. La razón está en que el registro de índice X se utiliza como desplazamiento variable, mientras que DESDE y HASTA son direcciones fuente y destino fijas.

El ejemplo revela que, aunque la indexación sea un recurso poderoso, no da lugar necesariamente a programas eficaces debido a las limitaciones de direccionamiento que imponen ciertos microprocesadores. Una genuina indexación de tipo general exige disponer de un desplazamiento o dirección de 16 bits y de un registro de índice de la misma longitud.

No obstante, hay que insistir en que el Z80 resuelve el problema mediante instrucciones especializadas. Describiremos a continuación una transferencia de bloques de tipo general que se ejecuta con sólo cuatro instrucciones. Pero sugerimos al lector, antes de pasar a analizarla, que resuelva los siguientes ejercicios, que le familiarizarán todavía más con el Z80:

Ejercicio 5.1: *Redáctese el programa de transferencia del Z80 en la forma del propuesto para el 6502, es decir, suponiendo que el registro de índice contiene un desplazamiento. Supóngase que los bloques fuente y destino se encuentran en la página 0 y, por tanto, en las direcciones 0 a 256. Naturalmente, se supondrá también que el número de elementos de cada bloque es suficientemente pequeño como para que no solapen.*

Ejercicio 5.2: *Supóngase ahora que los bloques destino y fuente se encuentran en cualquier otro lugar de la memoria, aunque siempre dentro de la misma página. Escríbase de nuevo el programa con arreglo a esta nueva convención. (¿Hay alguna diferencia?, es decir, ¿desempeña la página 0 alguna función en el Z80?)*

Figura 5.8
Transferencia de bloques: mapa de la memoria.

RUTINA GENERAL DE TRANSFERENCIA DE BLOQUES (PARA MAS DE 256 ELEMENTOS)

La distribución de registros y el mapa de la memoria se recogen en la figura 5.8. El programa es el siguiente:

```
LD BC,CUENTA   NUMERO DE BYTES
LD DE,HASTA    DIRECCION DE DESTINO
LD HL,DESDE    DIRECCION DE PARTIDA
LDIR           TRANSFERENCIA DE TODOS
               LOS BYTES
```

etalles, salta a la vista que es
or. La razón está en que el
omo desplazamiento variable,
son direcciones fuente y desti-

la indexación sea un recurso
mente ■ programas eficaces,
reccionamiento que imponen
nuina indexación de tipo ge-
azamiento o dirección de 16
e la misma longitud.
en que el Z80 resuelve el
especializadas. Describiremos
■ bloques de tipo general que
ucciones. Pero sugerimos al
, que resuelva los siguientes
davia más con el Z80:

a de transferencia del Z80 en
502, es decir, suponiendo que
■ desplazamiento. Supóngase
se encuentran en la página ■
0 a 256. Naturalmente, se
de elementos de cada bloque
para que no solapen.

os bloques destino y fuente se
gar de la memoria, aunque
gina. Escríbase de nuevo el
a convención. (¿Hay alguna
a página 0 alguna función en

FERENCIA DE
ELEMENTOS)

el mapa de la memoria se
ma es el siguiente:

DE BYTES
ON DE DESTINO
ON DE PARTIDA
RENCIA DE TODOS
ES

Memoria utilizada: 11 bytes.
Tiempo: 21 ciclos por byte transferido.
La primera instrucción es:

LD BC,CUENTA

y carga el número de elementos que han de transferirse (un valor de 16 bits) en el par de registros BC. Las dos siguientes instrucciones inicializan el par de registros DE y HL, respectivamente:

LD DE,HASTA
LD HL,DESDE

Por último, la cuarta instrucción:

LDIR

ejecuta la transferencia completa.

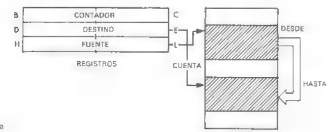


Figura ■■
Transferencia de bloques: mapa de la memoria.

LDIR es una instrucción *automática de transferencia de bloques*, con una potencia que este ejemplo hace obvia. LDIR da lugar a la siguiente secuencia: el contenido de la posición de memoria señalada por H y L se transfiere ■ la señalada por DE: (DE) = (HL); a continuación se incrementa DE: DE = DE + 1; luego HL: HL = HL + 1; por fin, se decrementa BC: BC = BC − 1; si BC vale 0, la instrucción termina; en caso contrario, se ejecuta de nuevo.

La utilidad y potencia de la instrucción LDIR debe ser ya evidente sin necesidad de más comentarios. De forma similar, la búsqueda del carácter "asterisco" puede llevarse ■ cabo con otra instrucción automática llamada CPIR, también especial del Z80.

El programa sería:

```
              LDA,"*"
              LD  BC,CUENTA
              LD  HL,SERIE
              CPIR
              JR  Z,ASTER
NOAST         ---
```

La primera instrucción carga el acumulador con el código del carácter asterisco. A continuación se inicializa el par de registros BC con la cuenta del número de palabras que hay que investigar dentro del bloque:

```
LD   BC,CUENTA
```

El par de registros H y L se lleva a la dirección de partida del bloque (SERIE). A continuación se ejecuta la instrucción automática:

```
LD   HL,SERIE
CPIR
```

La instrucción CPIR es una instrucción de comparación automática que compara el contenido de la posición de memoria especificada por HL con el del acumulador. Si la comparación es negativa, la bandera Z del registro de estado se activa a 1, el par HL se incrementa y el par BC se decrementa. La instrucción se repite hasta que el par BC vale 0 o hasta que la comparación es positiva. Por tanto, una vez ejecutada la instrucción CPIR, es preciso verificar la bandera Z, para determinar si el resultado ha sido o no positivo (en un caso extremo, la instrucción puede recorrer hasta 64K palabras sin haber obtenido nada positivo); ésta es la finalidad de la última instrucción

```
JR   Z, ASTER
```

Ejercicio 5.3: *Escríbase de nuevo el programa anterior de manera que la búsqueda vaya hacia atrás. (Un consejo: utilícese la instrucción CPDR.) La transferencia del bloque debe continuar hasta localizar el elemento "*".*

Vamos a desarrollar ahora un programa que combine las características de los dos anteriores, es decir, que ejecute la transferencia de un bloque desde la posición DESDE hasta la posición HASTA y que, a la vez, se detenga automáticamente si

encuentra el carácter de escape "asterisco". El programa sería así:

```
         LD   BC, CUENTA
         LD   HL, DESDE
         LD   DE, HASTA
         LD   A, "*"        DELIMITADOR
                            (CARACTER DE
                            ESCAPE)
PROBAR   CP   (HL)          COMPARAR CON
                            EL CARACTER EN
                            MEMORIA
         JR   Z, FIN        TERMINAR SI SE
                            ENCUENTRA
         LDI                TRANSFERIR EL
                            CARACTER Y AC-
                            TUALIZAR APUN-
                            TADORES Y
                            CUENTA
         JP   PE, PROBAR    SEGUIR PROBAN-
                            DO, SALVO QUE
                            P/V INDIQUE
                            BC = 0
```

Las tres primeras instrucciones del programa se encargan de la habitual inicialización de los registros de cuenta y de los punteros fuente y destino:

```
LD   BC, CUENTA
LD   HL, DESDE
LD   DE, HASTA
```

El carácter asterisco se carga en el acumulador en la forma habitual, de manera que pueda compararse con el carácter leído en la posición de memoria:

```
LD A, "*"
```

que es justamente lo que hace la siguiente instrucción:

```
PROBAR   CP   (HL)
```

El resultado positivo o negativo de la comparación se averigua verificando la bandera Z, que valdrá 1 en el primer caso. De la verificación se encarga la instrucción

```
JR   Z, END
```

Viene ahora una instrucción *automática de transferencia*:

LDI

que transfiere el carácter y actualiza los punteros y la cuenta de una vez. LDI transfiere el contenido señalado por H y L a la posición de memoria señalada por D y E: (DE) = (HL). También incrementa DE y HL:

DE = DE + 1
HL = HL + 1

Para terminar, decrementa BC: BC = BC −1. La peculiaridad de la instrucción radica en que la bandera P/V se borra si BC se decrementa hasta "0", y se pone a 1 en caso contrario. Esta situación la comprueba explícitamente la última instrucción del programa para determinar si es preciso salir o continuar:

JP PE, PROBAR

SUMA DE DOS BLOQUES

Este nuevo programa sirve para sumar dos bloques elemento a elemento a partir de las direcciones BLQ1 y BLQ2. CUENTA es el número de elementos de cada uno de los bloques, que deben tener idéntica longitud. El programa es:

```
SUMBLQ  LD    IX, BLQ1
        LD    IY, BLQ2
        LD    B, CUENTA
        XOR   A
BUCLE   LD    A, (IX + 0)
        ADC   A, (IY + 0)
        LD    (IX), A
        DEC   IX
        DEC   IY
        DEC   [illegible]
        JR    NZ, BUCLE
```

Figura 5.9
Suma de dos bloques: BLQ1 = BLQ1 + BLQ2.

La figura 5.9 recoge la distribución de la memoria. El programa es bastante sencillo: el número de elementos que deben sumarse se carga en el registro de contador B y los dos registros de índice IX e IY se inicializan a sus valores BLQ1 y BLQ2:

```
SUMBLQ  LD    IX, BLQ1
        LD    IY, BLQ2
        LD    B, CUENTA
```

Figura 6.9
Suma de dos bloques: BLQ1 = BLQ1 + BLQ2.

Antes de la primera suma se borra el bit de acarreo:

```
XOR   A
```

Se carga el primer elemento en el acumulador:

```
BUCLE     LD     A,(IX + 0)
```

A continuación se le suma el elemento correspondiente de BLQ2:

```
ADC   A,(IY + 0)
```

Y el resultado pasa a BLQ1:

```
LD   (IX),A
```

Los dos apuntadores X e Y se decrementan:

```
DEC   IX
DEC   IY
```

Lo mismo que el registro contador:

```
DEC   B
```

Si dicho contador no es 0, se repite el bucle de suma:

```
JR   NZ,BUCLE
```

Ejercicio 5.4: *¿Sabría utilizar el programa anterior para ejecutar una suma de 32 bits?*

Ejercicio 5.5: *¿Y una suma de 64 bits?*

Ejercicio 5.6: *Modifique el programa de manera que el resultado pase a un nuevo bloque que empiece en la dirección BLQ3.*

Ejercicio 5.7: *Modifique el programa para que ejecute una resta en lugar de una suma.*

Ejercicio 5.8: *Modifique el programa en el sentido de que BLQ1 y BLQ2 constituyen las respectivas partes superiores de cada uno de los bloques en lugar de las inferiores (véase figura 5.10).*

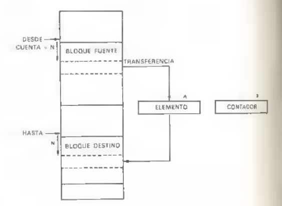


Figura 5.10
Organización de la memoria para transferencia de bloques.

Resumen

Hemos hecho una descripción completa de los modos de direccionamiento y hemos visto que el Z80 dispone de numerosos mecanismos y de modos de direccionamiento específicos. Los diversos programas de aplicaciones han servido para demostrar el valor de tales mecanismos. Quien desee aprender a programar con eficacia el Z80 deberá llegar a entenderlos perfectamente. A partir de ahora se utilizarán ampliamente en todos los programas del libro.

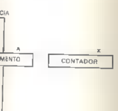
rograma anterior para ejecutar

bits?

na de manera que el resultado mpiece en la dirección BLQ3.

na para que ejecute una resta

a en el sentido de que BLQ1 y as partes superiores de cada e las inferiores (véase figura



completa de los modos de el Z80 dispone de numero- ireccionamiento específicos. ones han servido para de- s. Quien desee aprender a á llegar a entenderlos per- utilizarán ampliamente en

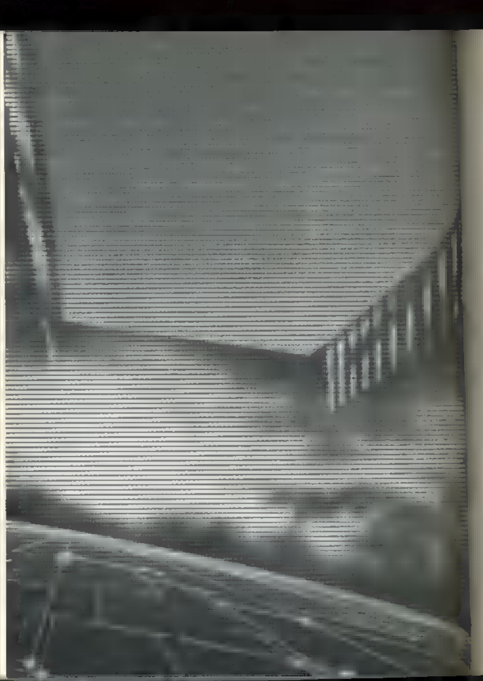
Ejercicio 5.9: *Escriba un programa para sumar los 10 primeros bytes de una tabla almacenada en la posición "BASE"; el resultado tendrá 16 bits (se trata de un cálculo de total de control).*

Ejercicio 5.10: *¿Podría resolver el mismo problema sin utilizar indexación?*

Ejercicio 5.11: *Invierta el orden de los 10 bytes de la tabla y almacene el resultado en la dirección "REVER".*

Ejercicio 5.12: *Busque el elemento mayor de esa misma tabla y almacénelo en la dirección de memoria "GRANDE".*

Ejercicio 5.13: *Sume los elementos correspondientes de las tres tablas, cuyas bases son BASE1, BASE2 y BASE3. La longitud de las tablas se almacena en la dirección "LONGITUD".*



de en

Introducción

Entrada/salid■

6

Técnicas de entrada/salida

Introducción

Hasta el momento hemos aprendido a intercambiar información entre la memoria y los diversos registros del procesador, a manipular dichos registros y a mover datos. Ahora debemos aprender a comunicarnos con el mundo exterior, que es justamente lo que se entiende por entrada/salida.

La *entrada* es la obtención de datos en los periféricos externos (teclado, discos o sensores físicos). *Salida* es la entrega de datos por parte del microprocesador o la memoria a dispositivos externos, como una impresora, una pantalla, un disco o sensores y relés.

En una primera etapa aprenderemos a realizar las operaciones de entrada/salida que exigen los dispositivos más comunes. En segundo lugar aprenderemos a manejar varios de esos dispositivos simultáneamente, es decir, a *organizarlos;* en esta segunda fase centraremos la discusión más en concreto en el uso del muestreo frente a las interrupciones.

Entrada/salida

Vamos antes de nada a aprender a percibir y a generar señales sencillas o impulsos. A continuación presentaremos algunas técnicas para garantizar o medir una sincronización

correcta. Una vez afianzada esta base, podremos pasar a estudiar técnicas de entrada/salida más complejas, como las transferencias a alta velocidad en serie y en paralelo.

INSTRUCCIONES DE ENTRADA/SALIDA DEL Z80

El Z80 dispone de una serie de instrucciones especiales para este cometido. La mayor parte de los microprocesadores de 8 bits carecen de ellas y controlan los dispositivos de entrada/salida con instrucciones generales. También el 8080 dispone de instrucciones especiales, aunque el Z80 cuenta con algunas nuevas, que describiremos más detalladamente para facilitar la comprensión de los programas que analizaremos en este capítulo.

Las instrucciones básicas de entrada y salida son, respectivamente: IN A, (n) y OUT (n), A, ambas heredadas del 8080. Leen o escriben, según, un byte entre la puerta seleccionada y el acumulador. El proceso de direccionamiento [illegible] tal que la dirección del dispositivo de E/S "n" se deposita en las líneas A0 a A7 del *bus* de direcciones, y el contenido del acumulador pasa a las líneas A8 a A15. Si sólo se direccionan 256 dispositivos, puede ser necesario llevar a 0 explícitamente el contenido del acumulador en el caso de que las líneas de dirección A8 a A15 puedan ser decodificadas por un dispositivo de E/S. En los sencillos ejemplos que veremos a continuación supondremos que hay menos de 256 dispositivos y que no están conectados a las direcciones que van de A8 [illegible] A15, de manera que no será preciso hacer 0 el contenido del acumulador explícitamente antes de emplear, por ejemplo, la instrucción IN.

Una instrucción de entrada especial —IN r,(C)— permite utilizar el contenido del registro C como dirección del dispositivo de E/S. Al utilizarla, el contenido del registro B proporciona automáticamente la parte superior de la dirección (A8 a A15). El registro especificado r se carga a partir de la dirección dada. "r" puede ser uno cualquiera de los siete registros de tipo general.

Figura 6.1
Conexión de un relé.

GENERACION DE UNA SEÑAL

En el caso más sencillo, es el ordenador el que desconecta (o conecta) los dispositivos de salida. Para modificar el estado de uno de tales dispositivos, el programador no tiene más que cambiar el nivel de un "0" lógico a un "1" lógico, o de "1" a "0". Supongamos que al bit "0" de un registro llamado "OUT1"

está conectado un relé externo; para conectarlo no tenemos más que escribir un "1" en la posición de bit adecuada del registro. Supongamos aquí que OUT1 representa la dirección de este registro de salida dentro de nuestro sistema; el siguiente programa conectaría el relé:

```
CONECTO   LD    A,00000001B    CARGAR DISPO-
                               SICION EN A
          OUT   (OUT1),A       ENVIARLA AL
                               DISPOSITIVO
```

donde OUT es la instrucción de salida.

Hemos supuesto que la situación de los siete bits restantes del registro OUT1 es irrelevante, aunque las cosas no suelen ser así, porque esos bits pueden estar conectados a otros relés; por tanto, vamos a mejorar tan elemental programa. Lo que haremos ahora es conectar el relé, pero sin alterar el estado de los demás bits del registro. Supongamos que es posible leer y escribir el contenido del mismo. La versión mejorada sería como sigue:

```
CONECTO   IN    A,(OUT1)       LEER EL CONTENI-
                               DO DE OUT1
          OR    00000001B      FORZAR BIT "0" A
                               "1" EN A
          OUT   (OUT1),A
```

El programa empieza por leer el contenido de la posición OUT1, al que a continuación somete a la operación OR, que pone a "1" el bit de posición 0 sin afectar al resto (para más detalles sobre la operación OR, consúltese el capítulo 4). La figura 6.1 recoge el proceso.

Figura 6.1
Conexión de un relé.

IMPULSOS

La generación de un *impulso* se lleva a cabo de la misma forma que acabamos de ver para el *nivel:* el bit de salida se pasa primero a 1, y a continuación otra vez a 0, lo que da lugar

a un impulso, como ilustra la figura 6.2. No obstante, en esta ocasión hay que resolver otro problema: el impulso debe durar un tiempo determinado. Por tanto, vamos a estudiar antes de nada cómo se produce un retraso en el ordenador.

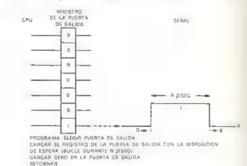


Figura 6.2
Impulso programado.

Figura 6.3
Diagrama de flujo del retardo básico.

PRODUCCION Y MEDIDA DE RETARDOS

El retardo puede producirse mediante los soportes lógico o físico. Aquí aprenderemos a crearlo en el programa, y más adelante ya estudiaremos la manera de obtenerlo con un contador físico o sincronizador de intervalos programable, como suele llamarse (PIT).

Los retardos programados se generan contando; el registro contador se carga con un valor que se va decrementando. El programa describe un bucle y reduce una y otra vez el contenido del contador hasta que llega a "0"; el tiempo consumido por esta serie de operaciones será el retardo buscado. Como ejemplo, vamos a generar un retardo de 82 ciclos de reloj:

```
RETARDO LD    A,5          A ES EL CONTA-
                           DOR
BUCLE   DEC   A            DECREMENTO
        JR    NZ,BUCLE     BUCLE COM-
                           PROBACION
```

El programa carga en A el valor 5; la siguiente instrucción decrementa A, y la otra provoca un salto a BUCLE mientras A no valga "0"; cuando alcanza este valor, el programa sale del bucle y ejecuta la instrucción que sigue. La lógica del programa es sencilla, y se ilustra en el diagrama de flujo de la figura 6.3.

ura 6.2. No obstante, en esta
blema: el impulso debe durar
o, vamos a estudiar antes de
o en el ordenador.





A CON LA DISPOSICION

RETARDOS

diante los soportes lógico o
o en el programa, y más
de obtenerlo con un conta-
rvalos programable, como

neran contando; el registro
se va decrementando. El
una y otra vez el conteni-
"; el tiempo consumido por
rdo buscado. Como ejem-
82 ciclos de reloj:

A ES EL CONTA-
DOR
DECREMENTO
BUCLE COM-
PROBACION

5; la siguiente instrucción
lto a BUCLE mientras A
lor, el programa sale del
e. La lógica del programa
de flujo de la figura 6.3.

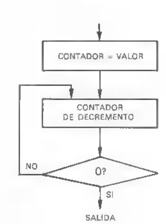


Figura 6.3
Diagrama de flujo del retardo básico.

Vamos ahora a calcular el retardo real provocado por el programa. En el capítulo 4 encontraremos el tiempo consumido por cada una de las instrucciones.

LD en modo inmediato necesita 7 ciclos; DEC emplea 4; por último, JR consume 12 ciclos, salvo en la última repetición, que sólo emplea 7. En efecto, al consultar la tabla correspondiente a JR vemos que hay dos posibilidades: si no hay salto, la instrucción termina en 7 ciclos, y si lo hay y debe recorrer el bucle, necesita 12.

Así que se consumen 7 ciclos en la primera instrucción, más 16 multiplicado por el número de veces que se repite el bucle en las dos siguientes menos 5, por el último salto que no se produce:

$$\text{Retardo} = 7 + 16 \times 5 - 5 = 82 \text{ ciclos.}$$

Si el ciclo es de 0.5 microsegundos, el retardo sería de 41 microsegundos.

El bucle de retardo que acabamos de examinar se emplea en casi todos los programas de entrada/salida; conviene, por tanto, entenderlo bien. Resuelva los dos ejercicios siguientes:

Ejercicio 6.1: *¿Cuáles son los retardos máximo y mínimo que pueden conseguirse con estas tres instrucciones?*

Ejercicio 6.2: *Modifíquese el programa para obtener un retardo de unos 100 microsegundos.*

Para conseguir un retardo más largo, una solución sencilla es añadir más instrucciones al programa antes de DEC; la instrucción más adecuada es NOP, que mantiene al procesador inactivo durante cuatro ciclos.

RETARDOS SUPERIORES

Para conseguir retardos mayores por medio del soporte lógico es necesario utilizar un contador más amplio; si éste tiene 16 bits, puede alojarse en un par de registros. Para simplificar, supongamos que la cuenta menor ■ "0". El byte inferior se carga con "0", y a continuación la superior recorre un bucle de decrementación. Como la primera operación reductora da lugar a 00 → FF y no afecta a la bandera Z cuando se decrementa a "0", el byte superior del contador se decrementará en 1. Cuando este byte llega al valor "0", el programa termina. Si hace falta una precisión mayor en la producción del retardo, la cuenta inferior puede llevar un valor no nulo. En este caso escribiríamos el programa tal como hemos explicado y añadiríamos al final el de tres líneas estudiado en el apartado anterior.

He aquí un programa de retardo de 24 bits:

RET24	LD	B, CUENTAS	CONTADOR SUPERIOR (8 BITS)
RET16	LD	D, − 1	
BUCLEA	LD	HL, CUENTAI	CONTADOR INFERIOR
BUCLEB	ADD	HL, DE	DECREMENTO HL
	JR	C, BUCLEB	SIGUE HASTA HACERLO NULO
	DJNZ	BUCLEA	DECREMENTO DE B Y SALTO

Obsérvese que DE se carga con "− 1", y se usa para decrementar el contador de 16 bits HL.

Naturalmente, pueden conseguirse retardos todavía mayores con más de tres palabras. El funcionamiento ■ análogo al de un cuentakilómetros de coche: cuando la rueda de la derecha pasa de 9 a 0, la siguiente se incrementa en 1. El cómputo con unidades discretas se basa siempre en este principio general.

El principal inconveniente de esta técnica es que el microprocesador pasa cientos de milisegundos y hasta de segundos

sin hacer nada. Si el ordenador no tiene ningún otro trabajo que realizar, esto no tiene importancia, pero, en general, deberá estar disponible para ejecutar otras tareas, y por eso los retardos muy largos no suelen producirse mediante el soporte lógico. De hecho, hasta los retardos más breves pueden ser problemáticos si el sistema debe garantizar una respuesta dentro de un plazo de tiempo limitado en algunas situaciones. En todos estos casos hay que recurrir a los retrasos de soporte físico. Por otra parte, cuando se trabaja con interrupciones, la detención por éstas de un bucle acarrearía la pérdida de la exactitud de la sincronización.

Ejercicio 6.3: *Escriba un programa para producir un retardo de 100 ms (típico de un teletipo).*

RETARDOS EN HARDWARE

Estos retardos se crean empleando un *sincronizador de intervalos programable* o cronómetro. Se carga un valor en el registro del cronómetro, y éste se encarga automáticamente de decrementar el contador. Por lo general, el programador puede ajustar o escoger el período; cuando llega a "0", el dispositivo suele enviar una interrupción al microprocesador; también puede activar un bit de estado inspeccionado periódicamente por el ordenador. El empleo de interrupciones se explicará más adelante en este mismo capítulo.

El cronómetro puede también partir de "0" y contar la duración de la señal o contar el número de impulsos recibido. Cuando funciona como cronómetro de intervalos, se dice que opera en modo *paso a paso*. Si cuenta impulsos, el modo se describe como *contador de impulsos*. Algunos dispositivos sincronizadores incluyen varios registros y recursos opcionales que el programador puede elegir.

RECEPCION DE IMPULSOS

El problema que plantea la recepción de impulsos es inverso al de su producción, pero con una dificultad adicional: mientras que la producción de un impulso tiene lugar bajo control del programa, su recepción es *asíncrona* respecto al mismo. Para detectarlo se utilizan dos técnicas: el *muestreo de dispositivos* y las *interrupciones:* de éstas hablaremos más adelante.

Detengámonos ahora en la técnica de muestreo, en virtud de la cual el programa lee continuamente el valor de un registro

dado de entrada y comprueba en el mismo un bit determinado (por ejemplo, el 0). Cada vez que se recibe un impulso, el bit adopta el valor "1"; el programa vigila el bit 0 hasta que pasa a "1", y en ese momento detecta un impulso. El programa necesario para ello es:

```
MUESTRA IN     A,(ENTRADA)   LEER REGISTRO
                             DE ENTRADA
ON      BIT    0,A           COMPROBAR EL
                             BIT 0
        JR     Z,MUESTRA     SEGUIR LA MUES-
                             TRA, SI ES 0
```

Supongamos ahora que la línea de entrada vale normalmente "1" y que deseamos detectar la presencia de "0". Es el procedimiento habitual para detectar un bit de ARRANQUE cuando se controla una línea conectada a un teletipo; el programa sería:

```
MUESTRA    IN     A,(ENTRADA)   LEER   REGIS-
                                TRO  DE  EN-
                                TRADA
           BIT    0,A           ACTIVAR   LA
                                BANDERA Z
           JR     NZ,MUESTRA    COMPROBAR
                                SI  SE  HA  IN-
                                VERTIDO
ARRANQUE ...
```

CONTROL DE LA DURACION

La duración de un impulso de entrada puede controlarse de la misma forma que se calcula la de uno de salida, y también en este caso puede recurrirse al soporte lógico o al físico. En el primer caso, lo normal es incrementar un contador de 1 en 1; para ello, el programa recorre un mismo bucle mientras hay impulso; cuando éste termina, su duración se calcula a partir del valor del registro contador. El programa es el siguiente:

```
DURACION  LD     B,0           BORRAR  CON-
                               TADOR
OTRA      IN     A,(ENTRADA)   LEER ENTRADA
          BIT    0,A           COMPROBAR
                               BIT 0
```


Transferencia

```
          JR    Z,OTRA          ESPERAR UN "1"
MAYOR     INC   B               INCREMENTAR
                                CONTADOR
          IN    A,(ENTRADA)     COMPROBAR
                                BIT 0
          BIT   0,A
          JR    NZ,MAYOR        ESPERAR UN "0"
```

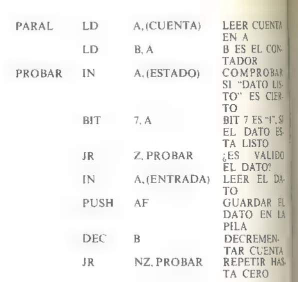
Naturalmente, se supone que la duración máxima del impulso no provocará el desbordamiento del registro B, porque en ese caso habría que modificar el programa, para tenerlo en cuenta (de no hacerlo así, se produciría un error).

Como ya sabemos producir y detectar impulsos, podemos pasar a recibir ■ transferir cantidades superiores de datos. En este nuevo problema cabe distinguir dos situaciones: transferencia en serie y transferencia en paralelo. Tras su solución empezaremos a trabajar con dispositivos de entrada/salida reales.

Transferencia de palabras en paralelo

Partimos de la suposición de que e■ la dirección "ENTRADA" (véase figura 6.4) disponemos en paralelo de 8 bits del dato a transferir. El microprocesador debe leer el byte situado en esa posición siempre que la palabra de estado le indique que es válida. Supondremos que dicha información de estado se encuentra en el bit 7 de la dirección "ESTADO". Vamos a escribir un programa que sea capaz de leer y reservar automáticamente cada una de las palabras de dato que reciba. Para simplificar las cosas, también supondremos que conocemos de antemano el número de palabras que deben leerse, almacenado en la posición "CUENTA". Si no dispusiésemos de tal información, tendríamos que buscar un *carácter de ruptura*, o *de borrado*, o quizá un asterisco "*", operación que ya sabemos hacer.

El diagrama de flujo aparece en la figura 6.5 y es muy sencillo. Se vigila la información de estado hasta que vale "1", lo que indica que una palabra está lista. Cuando ocurre esto, se lee y se guarda en una posición de memoria adecuada; a continuación se decrementa el contador y se comprueba si ha llegado a "0", lo que significaría el fin del programa; en caso contrario, se lee una nueva palabra. He aquí un programa sencillo que desarrolla el algoritmo descrito:



```
PARAL   LD    A,(CUENTA)    LEER CUENTA
                            EN A
        LD    B,A           B ES EL CON-
                            TADOR
PROBAR  IN    A,(ESTADO)    COMPROBAR
                            SI "DATO LIS-
                            TO" ES CIER-
                            TO
        BIT   7,A           BIT 7 ES "1", SI
                            EL DATO ES-
                            TA LISTO
        JR    Z,PROBAR      ¿ES VALIDO
                            EL DATO?
        IN    A,(ENTRADA)   LEER EL DA-
                            TO
        PUSH  AF            GUARDAR EL
                            DATO EN LA
                            PILA
        DEC   B             DECREMEN-
                            TAR CUENTA
        JR    NZ,PROBAR     REPETIR HAS-
                            TA CERO
```

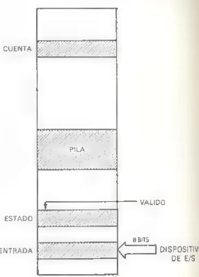


Figura 6.4
Transferencia de palabras en paralelo: memoria.

Figura 6.5
Transferencia de palabras en paralelo: diagrama de flujo.

JENTA) LEER CUENTA EN A
B ES EL CONTADOR
TADO) COMPROBAR SI "DATO LISTO" ES CIERTO
BIT 7 ES "1". SI EL DATO ESTA LISTO
DBAR ¿ES VALIDO EL DATO?
TRADA) LEER EL DATO
GUARDAR EL DATO EN LA PILA
DECREMENTAR CUENTA
OBAR REPETIR HASTA CERO



Se supone que la bandera "dato listo" se borra automáticamente al leer ESTADO.

Las dos primeras instrucciones inicializan el registro contador B:

```
PARAL    LD  A,(CUENTA)
         LD  B,A
```

Obsérvese que no hay forma fácil de cargar sólo B a partir de la memoria: es preciso cargar A y transferir su contenido a B o cargar B y C simultáneamente.

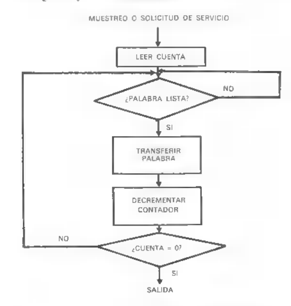


Figura 6.5
Transferencia de palabras en paralelo: diagrama de flujo.

Las tres instrucciones siguientes del programa leen la información de estado y recorren un bucle mientras el bit siete del registro de estado vale, "0" (se trata del bit del signo, es decir, del Bit N).

```
IN    A,ESTADO
BIT   7,A          "IN" NO ACTIVA LAS BANDERAS
JR    Z,PROBAR
```

Cuando JR no se realiza, el dato es válido y puede leerse:

```
IN  A,(ENTRADA)
```

La palabra ya se ha leído en la dirección de (ENTRADA) en que estaba, y ahora debe guardarse. Suponiendo que hay suficiente sitio en la pila, podemos usar la instrucción

```
PUSH  AF
```

que archiva A (y F) en la pila. Si ésta está completa, o si hay que transferir un número elevado de palabras, no podremos llevar a cabo la operación de PUSH, y tendremos que transferir los datos a un área de memoria designada al efecto mediante una instrucción indexada, por ejemplo. Lo malo es que ello exigiría una instrucción adicional para incrementar o decrementar el registro de índice. La operación PUSH es más rápida (sólo 11 ciclos de reloj).

La palabra dato ya está leída y archivada. Sólo nos queda decrementar el contador de palabra y comprobar si ha terminado:

```
DEC    B
JR     NZ, PROBAR
```

Ese programa de nueva instrucción podría considerarse de *referencia*. Se llama así a un programa cuidadosamente optimizado pensado para poner a prueba el rendimiento de un procesador ante una situación dada. La transferencia en paralelo es una de esas situaciones típicas, y el programa en cuestión está diseñado de manera que alcance la eficacia y la velocidad máximas. Vamos ahora a calcular la velocidad máxima de transferencia del mismo. Supondremos que CUENTA está en memoria. La duración de cada una de las instrucciones se determina consultando las tablas del capítulo 4; el resultado es el siguiente:

```
PARAL   LD      A,(CUENTA)      13
        LD      B,A              4
PROBAR  IN      A,(ESTADO)      11
        BIT     7,A              8
        JR      Z,PROBAR      7/12
        IN      A,(ENTRADA)     11
        PUSH    AF              11
        DEC     B                4
        JR      NZ,PROBAR     7/12
```

El tiempo mínimo de ejecución se obtiene suponiendo que cada vez que comprobamos ESTADO aparece un dato; en otras palabras, suponiendo que el primer salto JP falla siempre. En tal caso, el tiempo sería:

$$13 + 4 + (11 + 8 + 7 + 11 + 4 + 12) * \text{CUENTA}$$

Despreciando los primeros 17 ciclos necesarios para inicializar el registro contador, el tiempo necesario para transferir una palabra es de 64 ciclos de reloj, equivalentes a 32 microsegundos con un reloj de 2 MHz.

Por tanto, la cadencia de transferencia máxima es:

$$\frac{1}{32(10^{-4})} = 31\text{K por segundo}$$

Ejercicio 6.4: *Supongamos que han de transferirse más de 256 palabras. Modifique el programa en consecuencia y determine la influencia de la ampliación en la velocidad máxima de transferencia.*

Ejercicio 6.5: *Modifique el programa para tratar de aumentar su velocidad; siga uno de estos procedimientos:*

1. *Emplear JP en lugar de JR.*
2. *Utilizar DJNZ.*
3. *Utilizar INI o IND.*

¿Era el programa verdaderamente óptimo?

Ya sabemos hacer transferencia en paralelo a alta velocidad. Pasemos ahora a considerar un caso más complejo.

Transferencia de bits en serie

Se llama entrada en serie ■ la que tiene lugar de manera que los bits pasan, en secuencia, uno tras otro. Si llegan a intervalos regulares, se habla de transmisión *síncrona;* es *asíncrona* si la entrada se produce al azar, en forma de grupos de datos. Vamos ■ desarrollar un programa que funcione en los dos casos. El principio de la captación de datos secuenciales es sencillo: se comprueba una línea de entrada, que supondremos es la línea ∅; cuando se detecta un bit en la misma, se lee y se desplaza ■ un registro de almacenamiento; una vez reunidos 8 bits, el byte resultante se guarda en memoria, y se pasa ■ montar el siguiente. Para simplificar las cosas, supondremos que se sabe de antemano el número de bytes que va a recibirse; en caso contrario podríamos verificar la llegada de un carácter de

ruptura y detener la transferencia en ese momento, cosa que ya sabemos hacer. El diagrama de flujo del programa aparece en la figura 6.6; el programa es el siguiente:

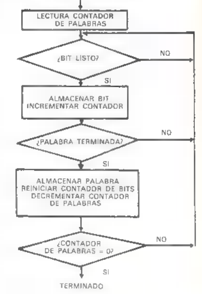


Figura 6.6
Transferencia de bits en serie: diagrama de flujo.

```
SERIE   LD    C,0           BORRAR PALABRA DE ENTRADA
        LD    A,(CUENTA)    CARGAR B CON EL CONTADOR DE BYTE
        LD    B,A
BUCLE   IN    A,ENTRADA     LEER PUERTA
        BIT   7,A           BIT 7 ES EL ESTADO Y BIT 0 EL DATO
        JR    Z,BUCLE       ESPERAR UN "1"
        SRL   A             DESPLAZAR EL BIT DE DATO AL ACARREO
```

Figura 6.7
Serie-Paralelo: Los registros.

en ese momento, cosa que ya
jo del programa aparece en la
uiente:

SERVICIO

NO

NO

NO

	BORRAR PALABRA DE ENTRADA
A)	CARGAR [redacted] CON EL CONTADOR DE BYTE
A	LEER PUERTA BIT 7 ES EL ESTADO Y BIT 0 EL DATO
	ESPERAR UN "1"
	DESPLAZAR EL BIT DE DATO AL ACARREO

RL	C	GUARDAR ENTRADA B EN C
JR	NC, BUCLE	SEGUIR HASTA QUE ENTREN 8 BITS
PUSH	BC	GUARDAR LA PALABRA EN LA PILA
LD	C,01H	LLEVAR A 0 EL BIT MARCADOR
DEC	[redacted]	DECREMENTAR EL CONTADOR DE BYTE
JR	NZ, BUCLE	MONTAR LA PALABRA SIGUIENTE

El programa está diseñado para conseguir la máxima eficacia y utiliza técnicas nuevas que explicaremos a continuación (véase figura 6.7).

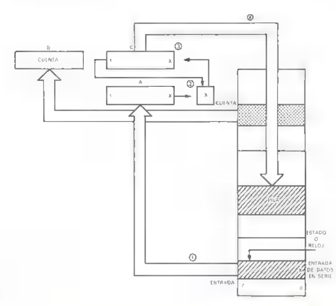


Figura 6.7
Serie-Paralelo: Los registros.

Las convenciones adoptadas son las siguientes: se supone que la posición de memoria CUENTA alberga el número de palabras que va ■ transferirse; el registro C se utiliza para reunir los 8 bits consecutivos que llegan; la dirección ENTRADA corresponde al registro de entrada; se supone que el bit 7 de este registro es una bandera de estado o un bit de reloj; el dato no es válido ■ es "0" y lo es ■ vale "1". El dato propiamente dicho se supone que aparece en el bit 0 de la misma dirección. En muchos casos, la información de estado no se encuentra en el registro de datos, sino en otros; por tanto, modificar el programa debe resultar sencillo. También se supone que el primer bit del dato recibido es siempre "1", lo que indica que a continuación viene el dato real. Si esto no fuese así, veremos más adelante una modificación obvia para tenerlo en cuenta. El programa responde exactamente al diagrama de flujo de la figura 6.6. Las primeras líneas del mismo realizan un bucle de espera que comprueba si hay un bit dispuesto; para determinar este hecho se lee el registro de entrada y a continuación se comprueba el bit 0 (Z); si éste vale "0", la instrucción JR se efectuará, y el programa entrará en el bucle. Cuando el bit de estado (o de reloj) indica certeza ("1"), el salto no se efectúa, y pasa ■ ejecutarse la siguiente instrucción.

Esta secuencia inicial corresponde a la flecha 1 de la figura 6.7.

En este punto el acumulador contiene un "1" en el bit 7 y el bit correspondiente al dato real en el bit 0. El primer bit que llega es "1", pero los siguientes pueden valer "0" o "1". Ahora nos interesa conservar el bit de dato recogido en la posición 0; la instrucción

```
SRL A
```

desplaza el contenido del acumulador una posición a la derecha, lo que hace que el bit derecho de A, que es el correspondiente al dato, pase al bit de acarreo. Ahora lo mantendremos en el registro C (el proceso está representado por las flechas 2 y 3 de la figura 6.7):

```
RL C
```

El efecto de esta instrucción es leer el bit de acarreo en la posición derecha de C. Al mismo tiempo, el bit de la izquierda de C pasa al bit de acarreo (si tiene alguna duda sobre la operación de rotación, consulte el capítulo 4).

Es importante recordar que la rotación con acarreo guarda el bit de acarreo, en este caso en la posición derecha, y también lo repone con el valor del bit 7 (o bit 0).

En nuestro ejemplo pasará un "0" al acarreo. La siguiente instrucción:

```
JR   NC, BUCLE
```

comprueba el acarreo y bifurca el programa ■ la dirección BUCLE, si aquél es "0"; es nuestro contador de bit automático. Es fácil comprobar que, como resultado de la primera aplicación de RL, C contendrá "00000001". Ocho desplazamientos más tarde, el "1" pasará, por fin, al bit de acarreo y detendrá la bifurcación. Es un método ingenioso de crear un contador automático de bucle sin desperdiciar una instrucción para decrementar el contenido del registro de índice; la técnica se emplea aquí para acortar el programa y mejorar su rendimiento.

Cuando deje de actuar JR NC, en C se habrán reunido 8 bits, valor que deberá conservarse en la memoria. De ello se encarga la siguiente instrucción (flecha 4 de la figura 6.7):

```
PUSH   BC
```

Los contenidos de B y C pasan a la pila; esta operación sólo es posible si la mencionada pila tiene sitio suficiente; suponiendo que tal condición se cumple, es la forma más rápida de guardar una palabra en memoria, a pesar de que también se guarda un registro innecesario (B); el apuntador de la pila se actualiza automáticamente. Si no queremos introducir la palabra en la pila, tendremos que utilizar una instrucción más para actualizar el puntero de la memoria. También podríamos emplear una dirección indexada equivalente, pero eso supondría decrementar o incrementar el índice, operación que consume todavía más tiempo.

Una vez almacenada la primera palabra del dato, ya no hay garantía de que el primer bit vuelva a ser "1"; por tanto, es preciso reiniciar el contenido a "00000001" para poder seguir utilizándolo como contador de bit; de ello se encarga la instrucción

```
LD   C,01H
```

Por último, decrementaremos el contador de palabras, ya que se ha montado una, y comprobaremos ■ hemos llegado al

término de la transferencia. Para ello sirven las dos instrucciones que siguen:

```
DEC    B
JR     NZ, BUCLE
```

El programa está diseñado en función de la velocidad, para que pueda captar una corriente rápida de bits de entrada. Cuando termina es aconsejable leer inmediatamente las palabras guardadas en la pila y transferirlas a cualquier otra posición de la memoria; en el capítulo 2 ya aprendimos a realizar una transferencia de bloques de esa naturaleza.

Ejercicio 6.6: *Calcúlese la máxima velocidad de lectura de bits en serie por el programa. Para ello se consulta en las tablas del capítulo 4 los ciclos que consume cada instrucción; para determinar el tiempo consumido por un bucle no hay más que multiplicar la duración total del mismo expresada en microsegundos por el número de veces que debe ejecutarse. A efectos de calcular la velocidad máxima, supóngase que hay un bit de dato listo cada vez que se detecta la posición de entrada.*

Este programa es más difícil de entender que los anteriores. Vamos, pues, a analizarlo de nuevo (consulte la figura 6.6) más en detalle, estudiando posibles alternativas.

De vez en cuando llega un bit de dato a la posición 0 de "ENTRADA"; puede haber, por ejemplo, tres "1" seguidos; por tanto, es preciso *diferenciar los bits sucesivos*. Esta es justamente la función de la señal de "reloj".

La señal de reloj (o ESTADO) dice que el bit de entrada es en este momento válido. Así pues, antes de leer un bit debemos verificar el de estado; si es "0", esperaremos; si es "1", significa que el bit de dato es correcto.

Hemos supuesto que la señal de estado está conectada al bit 7 del registro ENTRADA.

Ejercicio 6.7: *¿Puede explicar por qué se emplea el bit 7 para el estado y el 0 para el dato? ¿Tiene la elección alguna importancia?*

Una vez captado un bit, es necesario guardarlo en una posición segura, y a continuación desplazarlo a la izquierda, para dejar sitio al bit siguiente.

Por desgracia, el acumulador se está usando para leer y comprobar los datos y el estado; por tanto, si lo utilizásemos también para acumular el dato, el bit de la posición 7 podría ser borrado por el bit de estado.

Ejercicio 6.8: ¿Podría sugerir alguna forma de comprobar el bit de estado sin borrar el contenido del acumulador? ¿Sería con alguna instrucción especial? Si tal cosa fuese factible, ¿serviría el acumulador para conservar los bits que llegan en sucesión? ¿Aumentaría la velocidad con un salto automatizado?

Ejercicio 6.9: Vuelva a escribir el programa usando el acumulador para almacenar los bits de entrada. Compárelo con el anterior en términos de velocidad y número de instrucciones.

Vamos a examinar otras dos posibles variantes. En nuestro ejemplo hemos supuesto que el primer bit sería una señal especial y valdría siempre "1"; pero, en general, puede ser cualquiera.

Ejercicio 6.10: Modifique el programa anterior suponiendo que el primer bit es un dato válido que no debe descartarse y que puede valer tanto "0" como "1". Un consejo: el contador de bits seguirá funcionando correctamente si se inicializa el valor adecuado.

Para ganar tiempo, hemos guardado la palabra formada en la pila: pero, naturalmente, podríamos archivarla en cualquier posición especificada de la memoria.

Ejercicio 6.11: Modifique el programa anterior en el sentido de guardar la palabra reunida en el área de memoria que empieza en BASE.

Ejercicio 6.12: Modifique el programa anterior para que la transferencia se detenga en cuanto se detecte el carácter "S" en la corriente de entrada.

LA OPCION DEL SOPORTE FISICO

La mayor parte de los algoritmos de entrada/salida pueden ejecutarse en el soporte físico por medio de una pastilla llamada UART, que acumula automáticamente los bits. No obstante, si se desea reducir el gasto de componentes, puede utilizarse el programa que acabamos de ver o una variante del mismo.

Ejercicio 6.13: Modifique el programa en el sentido de suponer que el dato queda disponible en el bit 0 de la posición ENTRADA y la información de estado en el bit 0 de la dirección ENTRADA + 1.

Resumen básico de E/S

Ya sabemos realizar operaciones elementales de entrada/salida y manipular corrientes de datos en paralelo [redacted] en serie. Ahora ya podemos pensar en comunicarnos con dispositivos reales de entrada/salida.

Comunicación con dispositivos de entrada/salida

Para intercambiar datos con dispositivos de entrada/salida, el primer paso es asegurarse de que existen datos, [redacted] queremos leerlos, o de que el dispositivo está listo para aceptarlos, en el caso contrario. Para ello se recurre a dos técnicas: el acoplamiento y la interrupción; empecemos por la primera.

ACOPLAMIENTO

El acoplamiento es la técnica empleada habitualmente para establecer comunicación entre dos dispositivos asíncronos, es decir, no sincronizados. Si, por ejemplo, queremos enviar una palabra a una impresora en paralelo, antes de nada debemos asegurarnos de que está disponible la memoria auxiliar de entrada de dicho dispositivo. Así que preguntaremos [redacted] la impresora: "¿estás lista?", y ella responderá "sí" o "no". Si no está lista, habrá que esperar; [redacted] lo está, enviaremos los datos (véase figura 6.8).

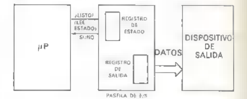


Figura 6.8
Acoplamiento (salida).

Y viceversa, antes de leer los datos procedentes de un dispositivo de entrada verificaremos si son válidos. Preguntaremos: "¿es válido el dato?", a lo que el dispositivo responderá "sí" o "no"; esta respuesta puede cifrarse por medio de los bits de estado [redacted] de alguna otra forma (véase figura 6.9).

Figura 6.9
Acoplamiento (entrada).

es elementales de entrada/sali-
atos en paralelo o en serie.
omunicarnos con dispositivos

entrada/salida

ispositivos de entrada/salida,
ue existen datos, [redacted] queremos
á listo para aceptarlos, en el
re a dos técnicas: el acopla-
nos por la primera.

npleada habitualmente para
dispositivos asíncronos, es
mplo, queremos enviar una
elo, antes de nada debemos
le la memoria auxiliar de
í que preguntaremos a la
sponderá "sí" o "no". Si no
está, enviaremos los datos



datos procedentes de un
[redacted] son válidos. Preguntare-
e el dispositivo responderá
arse por medio de los bits
(véase figura 6.9).

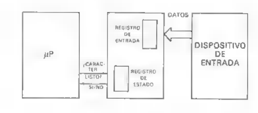


Figura [redacted]
Acoplamiento (entrada).

El acoplamiento se denomina también "apretón de manos", por analogía con la forma habitual impuesta por la cortesía de intercambiar información con una persona con la que no se tiene relación frecuente: antes de preguntarle nada, se le saluda y se le tiende la mano. El procedimiento de comunicación con dispositivos de entrada/salida sigue un ritual parecido, que ilustraremos a continuación con un ejemplo sencillo.

ENVIO DE UN CARACTER A LA IMPRESORA

Supondremos que el carácter se encuentra en la posición de memoria CAR. El programa de impresión es el siguiente:

```
ESPERA   IN    A,(ESTADO)
         BIT   7,A           COMPROBAR SI
                             ESTA LISTA
         JR    Z,ESPERA      EN CASO CON-
                             TRARIO, ESPE-
                             RAR
         LD    A,(CAR)       TOMAR EL CA-
                             RACTER
         OUT   (PRINTR),A    IMPRIMIRLO
         JR    ESPERA        IR A POR EL SI-
                             GUIENTE
```

El programa es bastante fácil de comprender y utiliza el procedimiento de contacto expuesto en la sección anterior. La figura 6.10 recoge las trayectorias de los datos.

El carácter (llamado DATO) se encuentra en la posición de memoria CAR. En primer lugar, se comprueba el estado de la impresora. Si el bit 7 del registro de estado se convierte en 1, [redacted] que el dispositivo está listo para aceptar la entrada, es decir, que su memoria auxiliar está disponible. En ese momento se carga el carácter en el acumulador, desde donde se envía a la salida. Mientras el bit de estado sea 0, el programa continuará describiendo el bucle del programa llamado ESPERA.

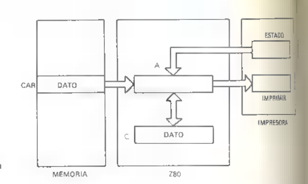


Figura 6.10
Trayectoria de los datos hacia la impresora.

Ejercicio 6.14: *¿Cuántas instrucciones podrían ahorrarse en el programa anterior cargando el dato directamente en el registro C y llevándolo también directamente a la salida desde el mismo?*

Ejercicio 6.15: *Cuando se utiliza una impresora, casi siempre es necesario enviar una orden de arranque antes de usarla. Modifíquese el programa propuesto para generar dicha orden, suponiendo que se obtiene escribiendo un 1 en el bit 0 del registro ESTADO, que es bidireccional.*

Ejercicio 6.16: *Si no existiese la instrucción BIT, ¿podría sustituirse por otra en la línea 2 del programa? En caso afirmativo, explíquese la ventaja de utilizar BIT, si es que tiene alguna.*

Ejercicio 6.17: *Modifíquese el programa propuesto en el sentido de imprimir una serie de n caracteres, siendo n menor que 255.*

Ejercicio 6.18: *Modifíquese el programa propuesto en el sentido de imprimir una serie de caracteres que debe interrumpirse cuando se encuentre un código de "retorno de carro".*

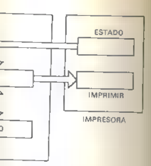
Vamos ahora a complicar el procedimiento de salida exigiendo un código de conversión y dirigiendo los datos simultáneamente a varios dispositivos.

SALIDA A UNA PANTALLA LED DE SIETE SEGMENTOS

Estas pantallas de diodos luminosos (LED) presentan las cifras "0" a "9" ó "0" a "F" hexadecimales iluminando diferentes combinaciones de un conjunto de 7 segmentos, como el que

Figura 6.11
LED de siete segmentos.

Figura 6.12
Caracteres hexadecimales representados con LED de siete segmentos.



...nes podrían ahorrarse en el ...dato directamente en el regis- ...ctamente a la salida desde el

...na impresora, casi siempre es ... arranque antes de usarla. ...sto para generar dicha orden, ...biendo un 1 en el bit 0 del ...ccional.

...nstrucción BIT, ¿podría sus- ...programa? En caso afirmativo, ... BIT, si es que tiene alguna.

...ama propuesto en el sentido ...eres, siendo n menor que 255.

...ama propuesto en el sentido ...eres que debe interrumpirse ...de "retorno de carro".

...rocedimiento de salida exi- ...irigiendo los datos simultá-

...ED DE SIETE

...nosos (LED) presentan las ...nales iluminando diferentes ...7 segmentos, como el que

aparece en la figura 6.11. La 6.12 recoge los caracteres que pueden generarse en una pantalla de este tipo.

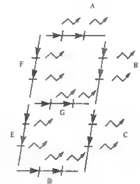


Figura 6.11
[redacted] de siete segmentos.

Los segmentos del LED se identifican mediante las letras "a" a "g" (véase figura 6.11). Así, para representar "0" se iluminan los segmentos abcdef. Supongamos ahora que el bit 0 de una puerta se conecta al segmento "a"; el 1, al "b"; etc.; el bit 7 no se utiliza. En estas condiciones, el código binario necesario para iluminar fedcba (para representar el valor "0") será "0111111", que en hexadecimal equivale a "3F". Realice el ejercicio que se propone a continuación.

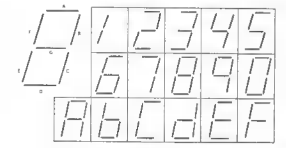


Figura 6.12
Caracteres hexadecimales representados con LED de siete segmentos.

Ejercicio 6.19: *Calcule los equivalentes en siete segmentos de las cifras hexadecimales "0" a "F", y anótelos en la tabla de abajo.*

Hex	*Cód. LED*	*Hex*	*Cód. LED*	*Hex*	*Cód. LED*	*Hex*	*Cód. LED*
0	3F	4		8		C	
1		5		9		D	
2		6		A		E	
3		7		B		F	

Pasemos ahora a la representación de valores hexadecimales en *varios* LED.

CONTROL DE VARIOS LED

Un LED no tiene memoria, y representa datos en tanto en cuanto sus segmentos estén activados por la corriente. Para que la pantalla de diodos resulte barata, el microprocesador representa la información *secuencialmente*, primero en un LED, luego en el siguiente, etc.; en evitación de parpadeo, la rotación de unos a otros debe ser rápida, inferior a 100 milisegundos. Vamos, pues, a diseñar un programa que satisfaga esta condición. Utilizaremos el registro C para señalar el punto [illegible] LED en que deseamos representar una cifra, cuyo valor hexadecimal estará contenido en el acumulador. El primer paso será convertir este valor hexadecimal en su representación equivalente en siete segmentos. En el último ejercicio hemos creado una tabla de equivalencias, a la que ahora accederemos con direccionamiento indexado, de manera que sea el propio valor hexadecimal el que proporcione el índice de desplazamiento. Así, el código de siete segmentos correspondiente a la cifra hexadecimal "3" se encuentra en el tercer elemento de la tabla a partir de la base: llamaremos a la dirección de ésta SEGBAS. El programa es:

```
LEDS    LD    E,A        A CONTIE-
                         NE UNA CI-
                         FRA HEX
        LD    D,0        UTILIZA
                         "DE" COMO
                         DESPLAZA-
                         MIENTO
```

```
         LD     HL,SEGBAS       UTILIZA
                                "HL" COMO
                                INDICE
         ADD    HL,DE           DIRECCION
                                DE LA TA-
                                BLA
         LD     A,(HL)          LEE UN CO-
                                DIGO EN
                                LA TABLA
         LD     B,50H           VALOR DE
                                RETRASO=
                                CUALQUIER
                                NUMERO
                                GRANDE
RETARDO  OUT    (C),A           SALIDA DU-
                                RANTE EL
                                TIEMPO FI-
                                JADO
         DEC    B               CONTADOR
                                DE RETRA-
                                SO
         JR     NZ,RETARDO      SIGUE EN
                                EL BUCLE
         LD     A,C             C ES EL NU-
                                MERO DE
                                PUERTA
         DEC    C
         CP     MINLED          ¿ESTA HE-
                                CHO EL UL-
                                TIMO LED?
         JR     NZ,OUT
         LD     BC,(MAXLED)     EN CASO
                                AFIRMATI-
                                VO, REINI-
                                CIAR C AL
                                PRIMERO
OUT      RET
```

El programa supone que el registro C contiene la dirección del LED que debe iluminarse a continuación, y que el acumulador A alberga la cifra que va a representarse.

Primero busca el código de siete segmentos correspondiente al valor hexadecimal contenido en el acumulador. Como campo de desplazamiento se usan los registros D y E; los H y L sirven

como registro de índice de 16 bits. La cifra hexadecimal se suma a la dirección de la base de la tabla:

```
LEDS    LD    E,A            CODIGO DE 7 SEGMENTOS
        LD    D,0
        LD    HL,SEGBAS
        ADD   HL,DE
```

A continuación se recorre un bucle de retardo para que el código obtenido en la tabla se represente durante un espacio de tiempo adecuado: en este caso se ha escogido arbitrariamente la constante hexadecimal "50":

```
LD  A,(HL)     LEE EL CODIGO EN LA TABLA
LD  B,50H      VALOR DE RETARDO
```

El retardo se obtiene mediante un bucle. La primera instrucción:

```
RETARDO  OUT  (C),A
```

saca el contenido del acumulador a la puerta de E/S señalada por el registro C (el número LED). Las siguientes dos instrucciones ejecutan el bucle de retardo:

```
DEC   B
JR    NZ,RETARDO
```

Al término del retardo no hay más que decrementar el puntero LED y adoptar las medidas necesarias para ir hasta la dirección LED más alta, si ya se ha pasado por la más baja:

```
      LD    A,C
      DEC   C
      CP    MINLED
      JR    NZ,OUT
      LD    BC,(MAXLED)
OUT   RET
```

Se supone que el programa es una subrutina, y por eso acaba con la instrucción RET (retorno de subrutina).

Ejercicio 6.20: *Por lo general, antes de representar una cifra es preciso desconectar los amplificadores de los LED. Modifíquese el programa anterior en consecuencia, añadiendo las ins-*

bits. La cifra hexadecimal se
de la tabla:

CODIGO DE 7
SEGMENTOS

AS

n bucle de retardo para que el
epresente durante un espacio de
ha escogido arbitrariamente la

CODIGO EN LA TABLA
DE RETARDO

e un bucle. La primera instruc-

r a la puerta de E/S señalada
D). Las siguientes dos instruc-
rdo:

más que decrementar el punte-
s necesarias para ir hasta la
e ha pasado por la más baja:

D)

es una subrutina, y por eso
etorno de subrutina).

es de representar una cifra es
adores de los LED. Modifique-
nsecuencia, añadiendo las ins-

trucciones necesarias (antes de llevar el carácter a la salida se lleva el código de carácter "00").

Ejercicio 6.21: *¿Qué ocurriría en la pantalla si se llevase retardo una línea más arriba? ¿Modificaría esto la duración? ¿Influiría en la representación de la pantalla?*

Ejercicio 6.22: *Como habrá observado, las primeras cuatro instrucciones del programa ejecutan, en realidad, un acceso indexado a la memoria de 16 bits que, debido a no utilizar el mecanismo de indexación, parece un tanto confuso. Supongamos que se conoce de antemano la dirección SEGBAS; llamemos SEGBSS a la parte superior de la misma y SEGBSI a la inferior, y almacenemos SEGBSS en la mitad superior del registro IX. Vuelva a escribir el programa propuesto utilizando el mecanismo de direccionamiento indexado del Z80 y SEGBSI como campo de desplazamiento de la instrucción. ¿Qué ventajas y qué inconvenientes derivan de este enfoque?*

Ejercicio 6.23: *El programa anterior es una subrutina, y, como habrá observado, utiliza internamente los registros B, D, E, H y L. Si la subrutina puede usar libremente las áreas de memoria designadas por las direcciones T1, T2, T3, T4 y T5, ¿sería capaz de añadir al principio y al final del programa las instrucciones necesarias para garantizar que, cuando la subrutina retorne, el contenido de los registros B, D, E, H y L sea el mismo que cuando entró?*

Ejercicio 6.24: *Repita el mismo ejercicio de antes, pero suponiendo que las áreas T1, T2, etc., no están disponibles para la subrutina. (Un consejo: recuerde que todos los ordenadores incorporan un mecanismo que conserva la información en orden cronológico.)*

Ya hemos resuelto varios problemas habituales de entrada/salida. Vamos a considerar el caso de un periférico típico: el teletipo.

ENTRADA/SALIDA POR TELETIPO

El teletipo es un dispositivo en serie que envía y recibe palabras de información en formato serie. Cada carácter se codifica en formato ASCII de 8 bits (la tabla ASCII se encuenra al final de este libro) y, además, va precedido de un bit de "arranque" y acabado por dos de "fin". En la conexión llamada de bucle de 20 miliamperios, que es la más común, el estado de la línea es normalmente "1", lo que indica al procesador que

dicha linea no se ha cortado. El arranque [redacted] una transición de "1" a "0", y señala al dispositivo receptor que a continuación empezarán a llegar bits de datos. El teletipo normal es un aparato de 10 caracteres por segundo. Acabamos de subrayar que cada carácter exige 11 bits, lo que significa que el teletipo debe transmitir 110 bits por segundo o bien 110 baudios. Vamos ahora [redacted] diseñar un programa que envíe datos [redacted] serie a un teletipo a la velocidad correcta.

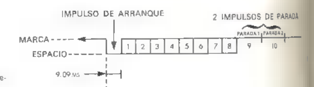


Figura 6.13
Formato de palabra en el teletipo.

Si se envían 110 bits por segundo, hay una separación entre bit y bit de 9,09 milisegundos, que debe ser la duración del bucle de retardo introducido entre bits sucesivos. El formato de la palabra en el teletipo aparece en la figura 6.13 y el diagrama de flujo de la entrada de bits en la 6.14. El programa es el siguiente:

```
TTIN     IN      A, ESTADO
         BIT     7, A           ¿DATOS LISTOS?
         JR      Z, TTIN        ESPERAR, EN CASO CONTRARIO
         CALL    RETARDO1       CENTRO DEL IMPULSO
         IN      A, (TTBIT)     BIT DE ARRANQUE
         OUT     (TTBIT), A     DEVOLVERLO
         CALL    RETARDO9       IMPULSO SIGUIENTE (9MS)
         LD      B, 08H         CONTADOR DE BITS
BUCLE    IN      A, (TTBIT)     LEER BIT DE DATO
         OUT     (TTBIT), A     DEVOLVERLO
         SRL     A              GUARDARLO EN EL ACARREO
```

Figura 6.14
Entrada a teletipo con devolución.

arranque es una transición de
o receptor que a continuación
os. El teletipo normal es un
gundo. Acabamos de subrayar
lo que significa que el teletipo
egundo o bien 110 baudios.
ama que envíe datos en serie a
cta.

2 IMPULSOS DE PARADA

PARADA 1 PARADA 2

5 6 7 8 9 10

ndo, hay una separación entre
que debe ser la duración del
bits sucesivos. El formato de
n la figura 6.13 y el diagrama
n la 6.14. El programa es el

DO

¿DATOS LISTOS?
ESPERAR, EN CASO CONTRARIO
OI CENTRO DEL IMPULSO
BIT DE ARRANQUE
DEVOLVERLO
O9 IMPULSO SIGUIENTE (9MS)
CONTADOR DE BITS
LEER BIT DE DATO
DEVOLVERLO
GUARDARLO EN EL ACARREO

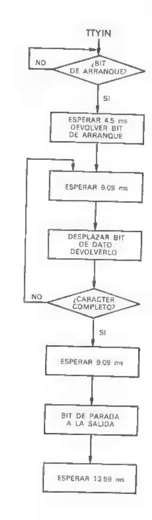


Figura 6.14
Entrada a teletipo con devolución

```
        RR      C               CONSERVARLO
                                EN C
        CALL    RETARDO9        IMPULSO SI-
                                GUIENTE (9MS)
        DEC     B               DECREMENTAR
                                EL CONTADOR
                                DE BITS
        JR      NZ, BUCLE
        IN      A,(TTBIT)       LEER BIT DE
                                PARADA
        OUT     (TTBIT),A       DEVOLVERLO
        CALL    RETARDO9        SALTAR AL SE-
                                GUNDO ALTO
        RET
```

Figura 6.15
Programa de teletipo.

Vamos a examinar el programa con cierto detalle. Lo primero es verificar el estado del teletipo para determinar si el carácter está disponible:

```
TTIN    IN      A, ESTADO
        BIT     7, A
        JR      Z, TTIN
```

La instrucción BIT es un recurso útil del Z80 que permite verificar cualquier bit de cualquier registro de datos sin modificar su contenido. La bandera Z se activa si el bit vale 0, y se pone a 0 en caso contrario.

El programa, por tanto, recorrerá un bucle hasta que el estado pase a "1"; es un bucle de barrido clásico.

Obsérvese que, como no es preciso mantener el contenido de ESTADO, también podría haberse usado la instrucción

```
AND     10000000B
```

en lugar de

```
BIT     7,A
```

aunque a costa de destruir el contenido de A (cosa que, en este caso, sería aceptable).

Al optimizar un programa, hay que tener en cuenta que cada instrucción nueva puede producir efectos secundarios.

A continuación se ejecuta un retardo de 4.5 ms para detectar el bit de arranque en el centro del impulso:

```
CALL RETARDO1
```

Figura 6.16
Entrada al teletipo.

RETARDO1 es la subrutina de retardo encargada de producirlo. El primer bit que llega es el de arranque, que debe devolverse al teletipo e ignorarse; de ello se encargan las instrucciones siguientes:

```
TTIN    IN      A,(TTBIT)
        OUT     (TTBIT),A
```

A continuación hay que esperar la llegada del primer bit de dato; el retardo necesario es de 9.09 milisegundos, y lo produce una subrutina:

```
CALL    RETARDO9
```

El registro B se usa como contador, y se carga con el valor 8 para captar datos de 8 bits:

```
LD   B,08H
```

A continuación los bits se leen uno por uno en el acumulador y se devuelven. Se supone que llegan en la posición 0 al acumulador, tras lo que se guardan en el registro C, donde se desplazan. La transferencia de A a C se lleva a cabo mediante el bit de acarreo:

```
BUCLE   IN      A,(TTBIT)
        OUT     (TTBIT),A
        SRL     A
        RR      C
```

La secuencia se ilustra en la figura 6.16.

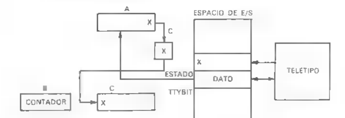


Figura 6.16
Entrada de teletipo.

A continuación se produce el retardo habitual de 9 milisegundos, se decrementa el contador de bits y se repite el bucle, si todavía no se han recibido los 8 bits:

```
CALL    RETARDO9
DEC     [illegible]
JR      NZ, BUCLE
```

Por último, se recoge y se devuelve el bit de parada. Por lo general, basta con uno, pero pueden enviarse dos con un par de instrucciones adicionales:

```
IN      A,(TTBIT)
OUT     (TTBIT),A
CALL    RETARDO9
RET
```

El programa debe estudiarse con atención. La lógica es bastante sencilla, y lo único nuevo es que todos los bits se leen en el teletipo (en la dirección TTBIT) y se devuelven al mismo. Esto es una característica propia de todos estos aparatos: cuando el usuario pulsa una tecla, la información se transmite al procesador, que la devuelve acto seguido a la impresora del teletipo. De esta forma se comprueba que las líneas de transmisión están en funcionamiento y que el procesador actúa correctamente.

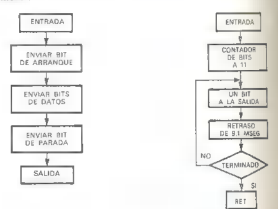


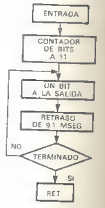
Figura 6.17
Salida del teletipo.

Ejercicio 6.25: *Escriba la rutina necesaria para producir un retraso de 9.09 milisegundos (subrutina RETARDO9).*

Ejercicio 6.26: *Utilizando como ejemplo el programa que acabamos de examinar, escriba otro PRINTC que imprima en el teletipo el contenido de la posición de memoria CAR (véase figura 6.16).*

...vuelve el bit de parada. Por lo ...den enviarse dos con un par de

... con atención. La lógica es ...o es que todos los bits se leen ...BIT) y se devuelven al mismo ...ía de todos estos aparatos; ... la información se transmite ...to seguido a la impresora del ...eba que las líneas de transmi... ...e el procesador actúa correc...



...necesaria para producir un ...brutina RETARDO9).

...plo el programa que acaba... ...PRINTC que imprima en el ...ón de memoria CAR (véase

He aquí la solución:

```
PRINTC  LD    B,11D        CONTADOR = 11 BITS
        LD    A,(CAR)      TOMAR EL CARACTER
        OR    A            BORRAR EL ACARREO = BIT DE ARRANQUE
        RLA                ACARREO EN A
BUCLE   OUT   (TTBIT),A    SALIDA
        CALL  RETARDO
        RRA                BIT SIGUIENTE
        SCF                ACARREO = 1 (BIT DE PARADA)
        DEC   B            CONTADOR DE BITS
        JR    NZ,BUCLE
        RET
```

El registro B se utiliza como contador de bits para la transmisión. El contenido del bit 0 de A se envía a la línea del teletipo ("TTBIT"). Obsérvese de qué forma se utiliza el acarreo para proporcionar el noveno bit (bit de ARRANQUE). Nótese también que el acarreo se borra mediante

```
OR   A
```

Al final del programa, el acarreo se activa a 1 mediante

```
SCF
```

para generar un bit de parada.

Ejercicio 6.27: *Modifique el programa para que espere un bit de ARRANQUE en lugar de un bit de ESTADO.*

IMPRESION DE UNA SERIE DE CARACTERES

Supondremos que la rutina PRINTC (véase ejercicio 6.26) se encarga de imprimir un carácter en la impresora o de presentarlo en cualquier dispositivo de salida. Vamos ahora a imprimir el

contenido situado entre las posiciones de memoria (PRIMER) (PRIMER + N). El programa es bastante simple (véase figura 6.18):

```
PSERIE   LD     B,NBR          LONGITUD
                               DE LA SERIE
         LD     HL,PRIMER      DIRECCION
                               DE BASE
BUCLE    LD     A,(HL)         TOMAR UN
                               CARACTER
         CALL   PRINTC         IMPRIMIRLO
         INC    HL             ELEMENTO
                               SIGUIENTE
         DEC    [redacted]
         JR     NZ,BUCLE       REPETIR
         RET
```

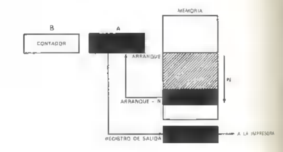
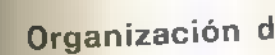


Figura 6.18
Impresión de un bloque de memoria.

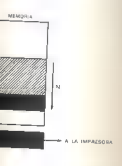
Resumen de periféricos

Ya hemos descrito las técnicas de programación básicas para comunicarse con dispositivos de entrada/salida típicos. Pero, además de transferir datos, es preciso preparar uno o más registros de control dentro de cada uno de los dispositivos de E/S para organizar las velocidades de transferencia, el mecanismo de interrupciones y otras varias posibilidades. A tal efecto es preciso consultar el manual que acompaña al dispositivo de que se trate (para más detalles sobre los algoritmos específicos de intercambio de información con todos los periféricos habituales, se remite al lector a la publicación *Microprocessor Interfacing Techniques*, de SYBEX, EE.UU., 2344 Sixth Street, Berkeley, California 94710.



ones de memoria (PRIMER y
bastante simple (véase figura

	LONGITUD DE LA SERIE
IMER	DIRECCION DE BASE
	TOMAR UN CARACTER
C	IMPRIMIRLO
	ELEMENTO SIGUIENTE
CLE	REPETIR



de programación básicas
de entrada/salida típicos.
preciso preparar uno o más
uno de los dispositivos de
e transferencia, el mecanis-
osibilidades. A tal efecto es
paña al dispositivo de que
algoritmos específicos de
s los periféricos habituales.
Microprocessor Interfacing
44 Sixth Street, Berkeley,

Sabemos manejar dispositivos individuales, pero en un sistema real todos están conectados a los *buses* y pueden solicitar atención simultáneamente. ¿Cómo se organiza el tiempo del procesador?

Organización de la entrada/salida

Como las solicitudes de entrada/salida pueden presentarse simultáneamente, es preciso crear un esquema de organización para determinar el orden en que se atenderán. Para ello se emplean tres técnicas básicas que pueden combinarse entre sí: muestreo de estaciones, interrupción, DMA. Describiremos aquí el muestreo y la interrupción, pero no el DMA, que es un dispositivo del soporte físico.

MUESTREO DE ESTACIONES

Desde el punto de vista conceptual, constituye la forma más fácil de manejar varios periféricos. El procesador interroga uno por uno a todos los dispositivos conectados a los *buses*; si uno de ellos solicita servicio, se lo proporciona; en caso contrario, se pasa a interrogar al periférico siguiente. La técnica es aplicable a *cualquier rutina de servicio de un dispositivo*.

Por ejemplo: si el sistema dispone de un teletipo, una grabadora de cinta magnética y una pantalla de rayos catódicos, la rutina de barrido preguntaría al teletipo: "¿tienes algún carácter que transmitir?"; a continuación se dirigiría a la *rutina de salida* del teletipo: "¿tienes que enviar algún carácter?" Si las dos respuestas fuesen negativas, interrogaría a las rutinas del aparato de cinta magnética y, por último, a la pantalla. Cuando sólo hay un dispositivo conectado al sistema, también puede usarse el muestreo para determinar si necesita servicio. Como ejemplo, las figuras 6.21 y 6.22 recogen los diagramas de flujo de lectura de una máquina lectora de cinta de papel y de impresión en una impresora.

El programa propuesto a continuación ejecuta un bucle de muestreo de los dispositivos 1, 2, 3 y 4 (véase figura 6.20):

MUESTREO4	IN	A, (ESTADO1)	ESTADO DEL DISPOSITIVO 1
	BIT	7, A	¿SOLICITUD DE SERVICIO?

```
CALL   NZ, UNO        ¿BIT 7 = 1?
IN     A, (ESTADO2)   DISPOSITIVO
                      2
BIT    7, A
CALL   NZ, DOS
IN     A, (ESTADO3)   DISPOSITIVO
                      3
BIT    7, A
CALL   NZ, TRES
IN     A, (ESTADO4)   DISPOSITIVO
                      4
BIT    7, A
CALL   NZ, CUATRO
JR     MUESTREO4      NO HAY RES-
                      PUESTA,
                      PROBAR DE
                      NUEVO
```

El bit 7 del registro de estado de cada dispositivo es "1", si desea servicio. Cuando detecta la solicitud, el programa salta al controlador del dispositivo, situado en la dirección UNO, para el 1; en la DOS, para el 2, etc.

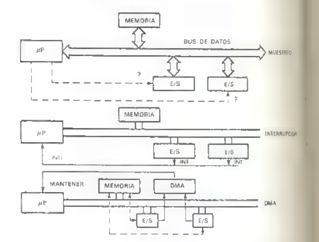


Figura 6.19
Tres métodos de control de E/S.

Hay un punto que conviene subrayar; es importante verificar de qué forma afectan a los códigos de condición cada una de las instrucciones. IN A no modifica las banderas, pero sí en

Figura 6.20
Diagrama de flujo del bucle del muestreo.

Figura 6.21
Lectura de una máquina lectora de cinta de papel.

su lugar se hubiese utilizado IN r, el bit 7 de la entrada reflejaría automáticamente el bit del SIGNO del registro de estado, y la instrucción BIT 7, A sería innecesaria; pero como IN A no cambia las banderas, es necesario incluir en el programa esa comprobación adicional.

En algunos soportes físicos, los dispositivos de entrada/salida pueden tratarse, a efectos de direccionamiento, como posiciones de memoria (entrada/salida por zona de memoria). En tal caso, la instrucción IN podría sustituirse por otra LD, dejando igual el resto del programa, porque LD no afecta a las banderas.

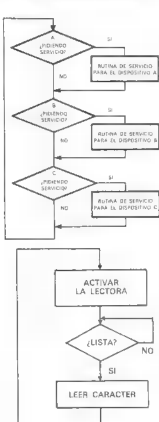


Figura 6.20
Diagrama de flujo del bucle del muestreo.

Figura 6.21
Lectura de una máquina lectora de cinta de papel.

Las ventajas del muestreo son obvias: es una técnica sencilla, no necesita apoyo del soporte físico y mantiene todas las entradas y salidas sincronizadas con el programa. El inconveniente también salta a la vista: la mayor parte del tiempo del procesador se pierde interrogando a dispositivos que no necesitan atención, lo que, entre otras cosas, podría hacer que llegase tarde a prestarlo a los que sí lo necesitan.

Por tanto, conviene disponer de otro mecanismo que deje al procesador libre para realizar cálculos útiles, en lugar de obligarle a interrogar continuamente a los periféricos. De todas formas, hay que insistir en que el muestreo se emplea muchísimo cuando el microprocesador no está muy sobrecargado, porque facilita mucho la organización del sistema. Estudiemos ahora la alternativa más importante al muestreo: la interrupción.

Figura 6.23
La pila del Z80 tras una interrupción.

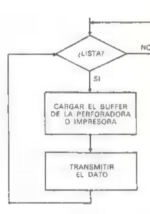


Figura 6.22
Impresión con una perforadora o una impresora.

Figura 6.24
Protección de algunos registros de trabajo.

INTERRUPCIONES

La idea de interrupción puede verse en la figura 6.19. Una línea especial del soporte físico, llamada línea de interrupciones, se conecta a una patilla específica del microprocesador. A esta línea pueden conectarse numerosos dispositivos de entrada/salida, y sirve para comunicar por medio de una señal que uno de ellos solicita atención al procesador. Veamos de qué forma responde éste a la solicitud.

En cualquier caso, el procesador termina de ejecutar la instrucción en curso, porque de otra forma se produciría un caos en su interior. A continuación salta a una rutina de

manipulación de interrupciones, que se encargará de llevar el caso; como se produce un salto, es necesario guardar en la pila el contenido del contador del programa; por tanto, *cualquier interrupción debe determinar automáticamente la conservación del contador del programa en la pila.* También hay que conservar automáticamente el registro de estado F, ya que su contenido se verá alterado por las instrucciones que sigan. Y, por último, será preciso preservar en la pila todos los registros internos susceptibles de ser modificados por la rutina (véanse figuras 6.23 y 6.24).

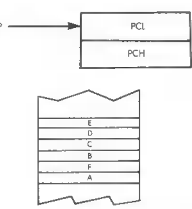


Figura 6.23
La pila del Z80 tras una interrupción

Figura 6.24
Protección de algunos registros de trabajo.

Una vez a salvo todos los registros que sean menester, puede saltarse a la dirección de manipulación de interrupciones. Al final de la rutina se restauran todos los registros y se ejecuta un retorno de interrupción especial, que tiene por objeto permitir la continuación del programa principal. Vamos a estudiar con más detalle la línea de interrupciones del Z80.

INTERRUPCIONES EN EL Z80

Una interrupción es una señal no sincronizada con el programa enviada al microprocesador en cualquier momento en solicitud de servicio. Cuando un programa salta a una subrutina, se dice que ésta está *sincronizada* con aquél, es decir, organizada por él; por el contrario, la interrupción puede producirse en cualquier momento y, por lo general, suspender la ejecución del programa en curso sin que éste lo "sepa".

Como puede producirse en cualquir momento, se dice que es *asíncrona*.

El Z80 dispone de tres mecanismos de interrupción: solicitud del *bus* (BUSRQ), interrupción no enmascarable (NMI) e interrupción normal (INT). A continuación estudiaremos los tres.

SOLICITUD DEL *BUS*

Es el mecanismo de interrupciones de máxima prioridad del Z80. La secuencia que desencadena aparece en la figura 6.25. Como norma general, el Z80 no percibirá ninguna interrupción hasta que no acabe con el ciclo de máquina en curso. Las interrupciones NMI e INT no se obedecen hasta que no termina la instrucción que se está ejecutando, pero BUSRQ se atiende en cuanto acaba el ciclo en curso, sin esperar a que termine la instrucción completa. Se utiliza para el acceso directo a la memoria (DMA), y hace que el Z80 pase a este modo. Si se ha llegado al término de una instrucción y hay pendientes señales NMI o INT, el Z80 las memoriza internamente, activando los correspondientes biestables especiales. En modo DMA, el procesador suspende todas las operaciones y pasa los *buses* de datos y direcciones al estado de alta impedancia; este modo es utilizado habitualmente por un controlador DMA para efectuar transferencias entre un dispositivo de entrada/salida de gran velocidad y la memoria, empleando para ello los *buses* de datos y direcciones. El término de la operación se indica mediante una serie de niveles cambiantes BUSRQ, momento en que el Z80 reanuda el funcionamiento normal, empezando por comprobar si están activados los biestables NMI o INT para atender las interrupciones correspondientes.

Salvo que la sincronización sea muy importante, el programador no tendrá que preocuparse por el DMA. Lo único que deberá tener en cuenta es que, si el sistema cuenta con un controlador de DMA, este modo puede retrasar la respuesta a las interrupciones NMI e INT.

Interrupción no enmascarable

Debe su nombre a que el programador no puede impedirla. El Z80 la acepta siempre al término de la instrucción en curso, salvo que haya recibido solicitud por *bus;* si NMI se recibe durante una operación BUSRQ, se activa el biestable interno NMI y se atiende al término de la instrucción que sigue a BUSRQ.

Figura 6.25
Secuencia de una interrupción

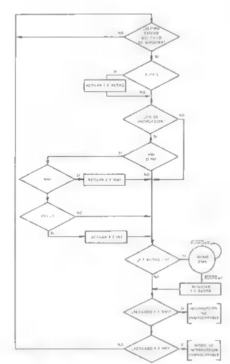


Figura 6.25
Secuencia de una interrupción.

NMI provoca el PUSH automático del contador del programa en la pila y el salto ■ la dirección 0066H, cuyos dos bytes pasan al contador mencionado: representan, pues, la dirección de partida de la rutina de manipulación de NMI (véase figura 6.26).

Este mecanismo de interrupción se utiliza en situaciones de "emergencia", y no ofrece la flexibilidad de la interrupción enmascarable que explicaremos más adelante.

Téngase en cuenta que hay que cargar una rutina de interrupciones en la dirección 0066H antes de usar NMI.

Cuando entra en acción, NMI desencadena la siguiente secuencia de operaciones:

$$\left.\begin{array}{l} SP \longleftarrow SP - 1 \\ (SP) \longleftarrow PCH \\ SP \longleftarrow SP - 1 \\ (SP) \longleftarrow PCL \end{array}\right\} \text{PUSH del PC}$$

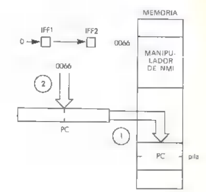


Figura 6.26
NMI provoca la vectorización automática.

Acto seguido, NMI provoca la reanudación automática en la posición 066H. La secuencia completa de acontecimientos es la siguiente:

PC ——→ PILA		(salvaguarda el contador del programa)
IFF1 ——→ IFF2		(salvaguarda IFF)
0 ——→ IFF1		(reinicia IFF)
SALTO A 0066H		(activa el manipulador de interrupciones)

El estado del biestable de interrupciones filtrables IFF1 presente en el momento de recepción de NMI se conserva automáticamente en IFF2, y a continuación se reinicia para evitar ulteriores interrupciones. Esto es importante para impedir la pérdida de la prioridad inferior de INT y simplfica el soporte físico externo, porque el estado de cualquier INT pendiente se conserva internamente en el Z80.

La interrupción NMI suele utilizarse en situaciones de alta prioridad, como reloj de tiempo real o fallo de la alimentación.

El retorno de NMI se encarga a una instrucción especial: RETN, retorno de una interrupción no enmascarable, en virtud de la cual se restaura el contenido de IFF1, a partir de IFF2, y el del contador del programa, a partir de su posición [redacted] la pila. Como IFF1 se había reiniciado durante la ejecución de NMI, durante ésta no puede haberse aceptado ninguna INT externa (salvo que el programador haya incluido una instrucción EI dentro de la rutina NMI); no ha habido, pues, pérdida de información.



reanudación automática en
pleta de acontecimientos es

lvaguarda el contador del
ograma)
lvaguarda IFF)
inicia IFF)
tiva el manipulador de in-
rupciones)

rupciones filtrables IFF1
ón de NMI se conserva
inuación se reinicia para
importante para impedir
NT y simplfica el soporte
alquier INT pendiente se

rse en situaciones de alta
fallo de la alimentación.
una instrucción especial:
enmascarable, en virtud
FF1, a partir de IFF2, y
de su posición en la pila,
te la ejecución de NMI,
do ninguna INT externa
uido una instrucción EI
abido, pues, pérdida de

Cuando acaba el manipulador de interrupciones, se produce la siguiente secuencia:

IFF2 ——► IFF1 (restaura el IFF)
PILA ——► PC (restaura el contador del programa)

Obsérvese que con el contenido de IFF1 se restaura la situación de admisión de interrupciones enmascarables.

Interrupción

La interrupción normal enmascarable INT puede actuar en tres modos, específicos del Z80, porque el 8080 sólo dispone de uno. El programador puede enmascarar selectivamente la interrupción normal INT: cuando los biestables IFF1 e IFF2 están en "1", el procesador recibe interrupciones; cuando se enmascaran a "0", las ignora. Estos biestables se activan a "1" mediante la instrucción EI, y a "0" mediante DI (los dos se activan y se reinician simultáneamente). Para evitar pérdidas de información, las interrupciones INT se ignoran durante la ejecución de las EI y DI. Pasemos ya a estudiar los tres modos de interrupción.

Modo de interrupción 0

Este modo es idéntico al de 8080. El Z80 opera en modo 0 cuando se pone en marcha inicialmente (cuando se aplica la señal RESET) o cuando se ejecuta una instrucción IM0. Una vez en este modo, se detecta una interrupción si el biestable IFF1 se encuentra en 1, siempre que no se produzca al mismo tiempo una solicitud del *bus* y una interrupción no enmascarable. La interrupción no se detecta hasta el término de la instrucción en curso. El Z80 responde a la misma generando una señal IORQ y otra M1, pasando a situación de espera sin hacer nada más.

Las señales IORQ y M1 debe detectarlas un *dispositivo externo* en una operación llamada *identificación de interrupción* o INTACK, y a continuación depositará una instrucción en el *bus* de datos. En el ciclo siguiente, el Z80 espera a que el dispositivo externo deposite en el *bus* dicha instrucción, que habitualmente es RST o CALL. Las dos guardan automáticamente el contador del programa en la pila y provocan el salto a una dirección determinada. La ventaja de RST es que ocupa un solo byte; por tanto, se ejecuta rápidamente; tiene el inconveniente de que sólo puede saltar a una de ocho posiciones de la página 0 (direcciones 0 a 255). La instrucción CALL tiene la ventaja de ser un salto de tipo general que especifica una dirección completa de 16 bits, aunque, como exige tres bytes, se ejecuta con más lentitud.

Obsérvese que se ignoran todas las interrupciones que pudieran llegar a partir del instante en que empieza a tratarse una, porque IFF1 e IFF2 pasan automáticamente a "0". A partir de este punto, el programador debe insertar una instrucción EI (que admite interrupciones) en la posición adecuada —y, en cualquier caso, antes del retorno de la interrupción en curso— si desea admitir otras adicionales.

La figura 6.27 recoge la secuencia que corresponde a una interrupción de modo 0.

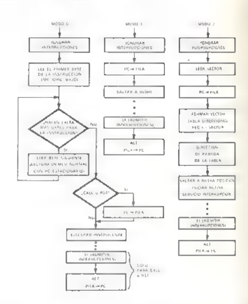


Figura 6.27
Modos de interrupción.

El retorno de una interrupción se efectúa mediante una instrucción RETI. Recordemos que el programador debe finalizar casi siempre explícitamente la interrupción que ha atendido al dispositivo de E/S, y siempre debe restaurar la bandera de bloqueo de interrupciones. No obstante, el controlador de periféricos puede utilizar la señal INTACK para acabar con la solicitud INT y liberar al programador de esa tarea.

Además, si la rutina de manipulación de interrupciones modifica el contenido de cualquiera de los registros internos, es responsabilidad exclusiva del programador guardarlos en la pila antes de ejecutar dicha rutina. En caso contrario, se destruirían

Figura 6.28
Guardar los registros.

los contenidos de tales registros, y cuando el programa reanudase el funcionamiento normal se produciría un fallo. Supongamos, por ejemplo, que el manipulador de interrupciones va a utilizar los registros A, B, C, D, E, H y L, el programador deberá guardarlos todos en la pila (véase figura 6.28).

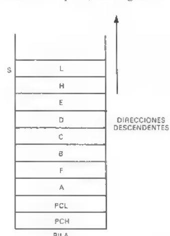


Figura 6.28
Guardar los registros.

El correspondiente programa es:

```
GUARDA  PUSH  AF
        PUSH  BC
        PUSH  DE
        PUSH  HL
```

Al término de la rutina de manipulación de interrupciones hay que restaurar todos los registros; el mencionado manipulador acabará con las instrucciones:

```
POP  HL
POP  DE
POP  BC
POP  AF
EI             (salvo que EI se hubiese utilizado en un punto anterior de la rutina)
```

También deben preservarse y restaurarse los registros IX e IY si los utiliza la rutina.

Modo de interrupción 1

Este modo entra en acción cuando se ejecuta la instrucción IM1. Es un manipulador automático de interrupciones que provoca el salto a la posición 0038H. Es, pues, una instrucción básicamente análoga a la interrupción NMI, con la diferencia de que es enmascarable. El Z80 guarda automáticamente el contenido del PC en la pila (véase figura 6.29).

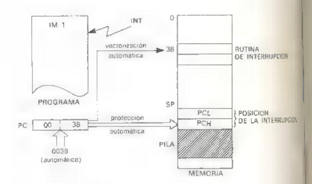


Figura 6.29
Modo de interrupción 1.

Esta respuesta automática que "vectoriza" todas las interrupciones a la dirección de memoria 38H tiene su origen en la necesidad del 8080 de reducir al mínimo la cantidad de soporte físico necesaria para trabajar con interrupciones. Su posible inconveniente es que salta *sólo* a una posición de memoria. Si hay varios dispositivos conectados a la línea INT, el programa que empiece en la dirección 38H se verá obligado a determinar cuál es el dispositivo que solicita atención; más adelante volveremos sobre este problema.

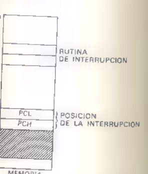
Hay que tener cuidado con la sincronización de esta interrupción, porque, al efectuar transferencias programadas de entrada y salida, el Z80 ignora todos los datos que pueda haber en el *bus* correspondiente durante el ciclo que sigue a la interrupción (ciclo de identificación de la interrupción).

Modo de interrupción 2 (interrupción vectorial)

Este modo se activa mediante la instrucción IM2. Es un recurso muy potente que permite la vectorización automática de interrupciones. El vector de interrupción es una dirección proporcionada por el dispositivo periférico que genera la parada, y que se usa como apuntador de memoria para la dirección de

Figura 6.30
Modo de interrupción 2.

ndo se ejecuta la instrucción
ático de interrupciones que
8H. Es, pues, una instrucción
ción NMI, con la diferencia
guarda automáticamente el
e figura 6.29).



"vectoriza" todas las inte-
a 38H tiene su origen en la
mo la cantidad de soporte
interrupciones. Su posible
a posición de memoria. Si
la línea INT, el programa
erá obligado a determinar
ión; más adelante volvere-

ncronización de esta inte-
ncias programadas de en-
os datos que pueda haber
ciclo que sigue a la inte-
interrupción).

vectorial)

instrucción IM2. Es un
ctorización automática de
ón es una dirección pro-
que genera la parada, y
oria para la dirección de

partida de la rutina de tratamiento de interrupciones. El mecanismo de direccionamiento porporcionado por el Z80 en modo 2 es indirecto: cada periférico suministra una dirección de salto de 7 bits que se añade a la de 8 bits contenida en el registro especial del Z80 I; el bit 0 (situado en el extremo derecho) de la dirección de 16 bits se pone ■ "0", lo que da lugar a otra dirección que apunta hacia la entrada a una tabla situada en cualquier lugar de la memoria. La tabla puede albergar hasta 128 entradas de palabras dobles, cada una de las cuales es la dirección del manipulador de interrupciones de un dispositivo. Las figuras 6.30 y 6.31 ilustran esta situación.

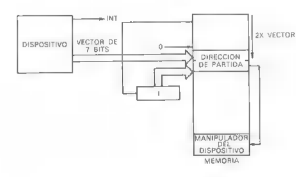


Figura 6.30
Modo ■ interrupción 2.

En este modo, el Z80 empuja automáticamente en la pila el contenido del contador del programa; es una precaución evidentemente necesaria, ya que el PC puede cargarse con el contenido de la entrada ■ la tabla de interrupciones correspondiente al vector proporcionado por el dispositivo.

Servicio de las interrupciones

En la figura 6.19 se hace una comparación gráfica entre el muestreo y la interrupción. Es fácil ver que la primera técnica obliga al programa a perder muchísimo tiempo tomando muestras.

El otro mecanismo detiene el programa, es atendido y deja que éste siga su curso normal. El inconveniente de la interrupción es que obliga a añadir varias instrucciones al principio y al final que retrasan la ejecución de la primera instrucción del manipulador de interrupciones; son los servicios generales de este mecanismo.

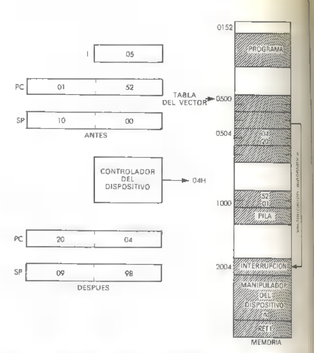


Figura 6.31
Un ejemplo práctico de modo 2.

Ejercicio 6.28: *Calcule, a partir de las tablas de ciclos por instrucción del capítulo 4, el tiempo que se pierde en guardar y restaurar los registros A, B, D y H.*

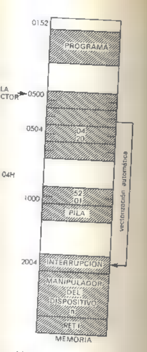
Una vez aclarado el funcionamiento de las líneas de interrupción, nos centraremos en dos importantes problemas todavía pendientes:

1. ¿Qué hacer cuando varios dispositivos solicitan una interrupción al mismo tiempo?
2. ¿Qué hacer si se presenta una interrupción mientras se atiende otra?

Figura 6.32
Interrupciones muestreadas vectorizadas.

VARIOS DISPOSITIVOS CONECTADOS A UNA MISMA LINEA DE INTERRUPCIONES

Cada vez que ocurre una interrupción, el procesador salta a una dirección determinada, pero, antes de que pueda hacer nada, la rutina de tratamiento de interrupciones debe determi-



tablas de ciclos por ins-
ue se pierde en guardar y
H.

to de las líneas de inte-
ortantes problemas toda-

ositivos solicitan una in-

interrupción mientras se

DOS A UNA
NES

n, el procesador salta a
s de que pueda hacer
upciones debe determi-

nar de qué dispositivo procede la señal. Como es habitual, la identificación puede hacerse mediante el soporte lógico o mediante el físico.

En el primer caso se recurre al muestreo; el microprocesador pregunta a un dispositivo: "¿has solicitado una interrupción?" Si la respuesta ■ negativa, dirige la pregunta al siguiente. La figura 6.32 recoge el método. El programa de muestreo es el siguiente:

MUESTRA	IN	A,(ESTADO)	LEE EL ESTADO
	BIT	7, A	¿HA SOLICITADO INTERRUPCION EL DISPOSITIVO?
	JP	NZ, UNO	EN CASO AFIRMATIVO, ATENDERLA
	IN	A,(ESTADO2)	
	BIT	7, A	
	JP	NZ, DOS	
	etc.	...	

El soporte físico proporciona la dirección del dispostivo, ■ la par que la solicitud de interrupción.

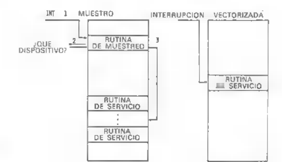


Figura 6.32
Interrupciones muestreadas y vectorizadas.

Para ser más exacto, cuando se opera en modo 0, el controlador del periférico deposita una instrucción RST de un byte o una CALL de tres bytes en el *bus* de datos como respuesta al reconocimiento de interrupción, lo que automatiza la vectorización de interrupciones y reduce el tiempo de servicio necesario.

Téngase en cuenta que hace falta una instrucción de llamada a subrutina, porque el Z80 no guarda el PC cuando funciona en modo 0.

En la mayor parte de los casos la velocidad de respuesta a una interrupción no es crucial, por lo que se emplea el mecanismo de muestreo. Pero si ese tiempo es crucial, hay que recurrir al soporte físico.

INTERRUPCIONES SIMULTANEAS

El siguiente problema es la solicitud de una interrupción durante la ejecución de una rutina de manipulación de interrupciones. Vamos a ver lo que ocurre y cómo puede usarse la pila para salir del paso. En el capítulo 2 ya adelantamos que esta función era una de las primordiales de la pila, y ahora vamos a demostrarlo. El problema general se ilustra en la figura 6.34, en la que el tiempo transcurre de izquierda a derecha; la parte inferior de la misma recoge el contenido de la pila. Empezando por la izquierda, en el momento T0 está en marcha el programa P. En T1 se produce la interrupción I1, que se autoriza. El programa P se detiene, como queda indicado en la parte inferior de la figura. La pila contiene, como mínimo, el contador del programa y el registro de estado de P, más cualesquiera otros registros guardados por el manipulador de interrupciones o por la propia I1.

Figura 6.34
El contenido de la pila durante una serie de interrupciones múltiples.



Figura 6.33
Varios dispositivos compartiendo una misma línea de interrupciones.

En T1 empieza a ejecutarse I1. En T2 se produce la interrupción I2, que en este ejemplo supondremos de prioridad superior a I1. Si hubiera tenido menos prioridad, se habría ignorado hasta la terminación de I1. En T2 se llevan a la pila los registros de I1, como indica la parte inferior de la ilustración; una vez más se guardan en la pila los contenidos del contador del programa y de AF. Además, la rutina de I2 podría proteger nuevos registros. A partir de este momento, la interrupción I2 se ejecuta hasta que termina en el momento T3.

Cuando acaba —con una instrucción RETI—, el contenido de la pila pasa automáticamente al Z80, como se ve en la parte inferior de la figura 6.34, de manera que se reanuda también automáticamente la ejecución de I1. Pero en T4 vuelve a produ-

Resumen



cirse una nueva interrupción I3 de mayor prioridad; los registros de I1 vuelven otra vez a la pila. La nueva interrupción se ejecuta entre T4 y T5, punto en el que concluye. El contenido de la pila pasa de nuevo al Z80, y prosigue la ejecución de I1, que en esta ocasión avanza sin más interrupciones hasta terminar en T6. En este momento, los registros que estaban guardados en la pila pasan al Z80, y continúa la ejecución del programa P. Como el lector puede comprobar, cuando esto ocurre, la pila ya está vacía (el número de líneas de puntos que señalan la interrupción del programa corresponde al de niveles almacenados en la pila).

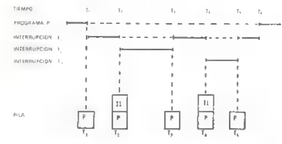


Figura 6.34
El contenido de la pila durante una serie de interrupciones múltiples.

Ejercicio 6.29: *Supongamos que el área a disposición de la pila en un programa determinado está limitado a 300 posiciones, que es preciso guardar todos los registros y que el programador permite la existencia de interrupciones internas, es decir, de interrupciones dentro de interrupciones. ¿Cuál será el máximo número de éstas admisible simultáneamente? ¿Hay algún otro factor capaz de reducir todavía más ese número?*

Hay que subrayar que los microprocesadores no suelen estar conectados a una cantidad grande de dispositivos generadores de interrupciones, por lo que la probabilidad de que se produzcan muchas simultáneamente es baja.

Ya hemos resuelto todos los problemas asociados habitualmente a las interrupciones. El mecanismo es fácil de manejar, y hasta el programador sin experiencia debe ser capaz de utilizarlo con aprovechamiento.

Resumen

Hemos pasado revista en este capítulo a todas las técnicas de comunicación con el mundo exterior. Desde las rutinas de entrada/salida más elementales hasta programas de comunica-

ción con periféricos reales, hemos aprendido a desarrollar todas las rutinas normales, y hasta hemos analizado la eficacia de un par de programas de referencia de transferencia en paralelo y de conversión paralelo a serie. La parte final del capítulo se ha dedicado a la organización del trabajo con varios periféricos mediante los mecanismos de muestreo e interrupción. A un sistema pueden conectarse muchos otros dispositivos de entrada/salida de la naturaleza más dispar, pero con las técnicas expuestas y si se comprende su funcionamiento, debe ser posible resolver la mayor parte de los problemas.

En el capítulo próximo veremos las características de las pastillas de conexión de entrada/salida que suelen acoplarse al Z80, para considerar a continuación las estructuras básicas de datos que puede emplear el programador.

Ejercicio 6.30: *Calcúlese la duración del servicio en modo 0 suponiendo que se guardan todos los registros y que se recibe una instrucción RST en respuesta a la detección de la interrupción. El servicio es el retraso total producido cuando se atiende una interrupción, con exclusión del debido a la ejecución de las instrucciones necesarias para tratar la interrupción propiamente dicha.*

Ejercicio 6.31: *Los LED de 7 segmentos no sólo sirven para representar las cifras del sistema hexadecimal. Determínense los códigos de representación de: H, I, J, L, O, P, S, U, Y, g, h, i, j, l, n, o, p, r, t, u, y.*

Ejercicio 6.32: *La figura 6.34 recoge el diagrama de flujo del tratamiento de una interrupción. Responda a las siguientes preguntas:*

a) *¿Qué hace el soporte lógico y qué el físico?*
b) *¿Para qué sirve el enmascaramiento?*
c) *¿Cuántos registros deben guardarse?*
d) *¿Cómo se identifica el dispositivo interruptor?*
e) *¿Qué hace la instrucción RETI? ¿En qué se diferencia de un retorno de subrutina?*
f) *¿Cómo podría solucionarse una situación de desbordamiento de la pila?*
g) *¿Qué tiempo de servicio ("tiempo perdido") introduce el mecanismo de interrupción?*

Figura 6.35
Lógica de una interrupción.

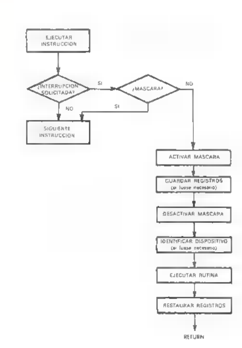
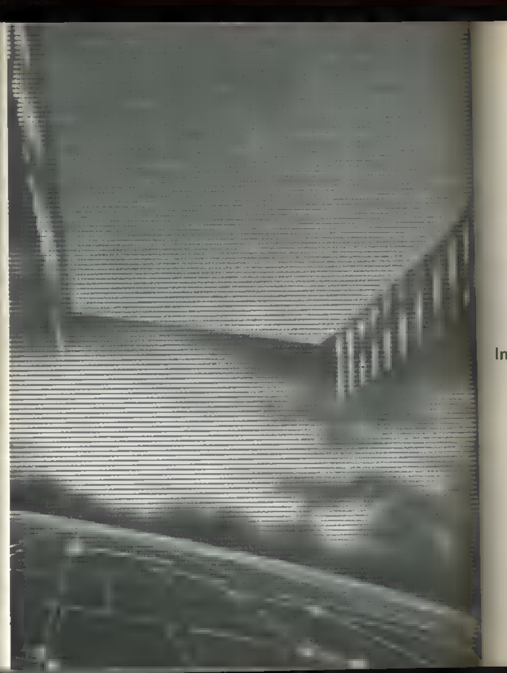


Figura **6.35**
Lógica de una interrupción.



Introducción

7

Dispositivos de entrada/salida

Introducción

Ya sabemos programar el microprocesador Z80 en la mayor parte de las situaciones habituales, pero es preciso mencionar las pastillas de entrada/salida conectadas normalmente al mismo. Los constantes progresos de integración LSI (integración a gran escala) han determinado la introducción de circuitos antes inexistentes, de manera que la programación de un sistema supone, en primer lugar, la del propio microprocesador y, en segundo lugar, la de las *pastillas de entrada/salida*. De hecho, muchas veces es más difícil recordar cómo deben programarse las diferentes opciones de control de las pastillas de E/S que programar el microprocesador, y no porque la programación en sí sea más complicada, sino porque cada uno de esos dispositivos tiene sus peculiaridades. Examinaremos aquí el tipo más general —el dispositivo de entrada/salida programable o PIO—, y a continuación veremos algunos dispositivos E/S de Zilog.

"PIO estándar"

No existe el "PIO estándar", pero todos los fabricantes de las distintas marcas con el mismo fin funcionan básicamente igual. La finalidad de un PIO es proporcionar una

conexión multipuerta a los dispositivos de entrada/salida (una puerta no es más que un juego de 8 líneas de entrada/salida). Cada PIO proporciona al menos dos juegos de líneas de 8 bits a los dispositivos de E/S. Estos necesitan siempre una *memoria auxiliar* para estabilizar el contenido del *bus* de datos, por lo menos a la salida. Los PIO dispondrán, por tanto, de al menos una memoria auxiliar por puerta.

Como ya se ha dicho, el microprocesador utiliza para comunicarse con los dispositivos de E/S un mecanismo de *acoplamiento* o de *interrupción*: los PIO se comunican con los periféricos de forma parecida, y para ello disponen de un mínimo de dos *líneas por puerta* para efectuar la función de acoplamiento.

El microprocesador necesita saber el estado de cada una de las puertas, para lo que, a tal fin, cuenta con uno o más *bits de estado*. Por último, cada PIO dispone de una serie de recursos que configuran sus posibilidades. Para especificar las opciones de programación, el usuario tiene que acceder a un registro especial interno del PIO llamado *registro de control;* en algunos casos, la información de estado forma parte de este registro.

Esencial a cualquier PIO es la posibilidad de configurar cada una de las líneas como de entrada o de salida. La figura 7.1 recoge el esquema de un PIO. Todas las puertas disponen de un *registro de dirección de datos* que sirve para programar la dirección de las líneas. En muchos PIO, un bit "0" en una posición de ese registro significa entrada, y un "1", salida. Zilog utiliza la convención contraria.

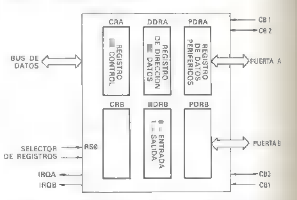


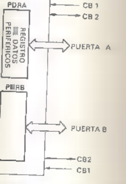
Figura 7.1
Un PIO típico.

Quizá resulte sorprendente utilizar "0" para la entrada y "1" para la salida, cuando, en realidad, es más natural adjudi-

sitivos de entrada/salida (una
de 8 líneas de entrada/salida).
dos juegos de líneas de 8 bits
ecesitan siempre una *memoria*
ido del *bus* de datos. por lo
ndrán, por tanto, de al menos
.

nicroprocesador utiliza para
de E/S un mecanismo de
s PIO se comunican con los
para ello disponen de un
para efectuar la función de

ber el estado de cada una de
uenta con uno ■ más *bits de*
one de una serie de recursos
Para especificar las opciones
que acceder ■ un registro
gistro de control; en algunos
rma parte de este registro.
a posibilidad de configurar
trada o de salida. La figura
Todas las puertas disponen
que sirve para programar la
s PIO, un bit "0" en una
rada, y un "1", salida. Zilog



ar "0" para la entrada y
d, es más natural adjudi-

car "0" a la salida (Output en inglés) y "1" a la entrada (Input); pero se trata de una elección deliberada: cuando se aplica al sistema la alimentación, es muy importante que todas las líneas de E/S estén en configuración de *entrada*, porque en caso contrario, si el microprocesador estuviese conectado a algún periférico peligroso, podría activarlo accidentalmente. Cuando se aplica la señal de reiniciación (reset), normalmente todos los registros pasan a 0, de manera que todas las líneas del PIO quedan configuradas como entradas. Las conexiones con el microprocesador están representadas a la izquierda de la figura mencionada. Como es natural, el PIO se conecta al *bus* de datos de 8 bits, al de direcciones y al de control del microprocesador. El programador no tiene más que especificar la dirección de cualquiera de los registros del PIO al que desee acceder.

REGISTRO INTERNO DE CONTROL

Este registro cuenta con una serie de opciones de generación y detección de interrupciones o de ejecución automática de la función de acoplamiento. No es necesario hacer aquí una descripción completa de sus recursos; el usuario del sistema no tiene más que consultar la hoja de características, que recoge el efecto de activar los diferentes bits del registro de control. Cada vez que se inicialice el sistema, el usuario deberá cargar el mencionado registro del PIO con el contenido adecuado a la aplicación en uso.

Programación de un PIO

La siguiente sería una secuencia típica de empleo de un canal de PIO (suponiendo una entrada):

Carga del registro de control

Se efectúa mediante una transferencia programada entre un registro del Z80 (por lo general, el acumulador) y el registro de control del PIO. De esta forma quedan activadas las opciones y el modo de funcionamiento del PIO (véase figura 7.2). Normalmente sólo se hace una vez, al principio del programa.

Carga del registro de dirección

Especifica la dirección en que deben utilizarse las líneas de E/S (véase figura 7.3).

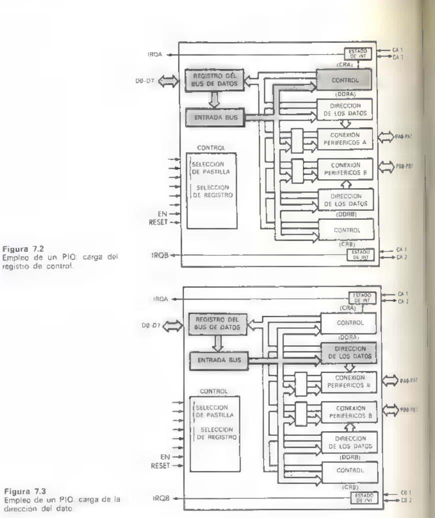


Figura 7.2
Empleo de un PIO: carga del registro de control.

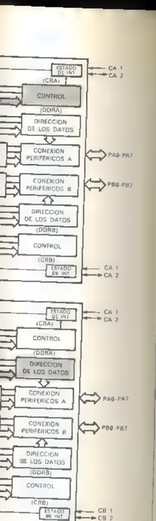
Figura 7.3
Empleo de un PIO: carga de la dirección del dato.

Lectura del estado

El registro de estado indica si hay o no un byte válido a la entrada (véase figura 7.4).

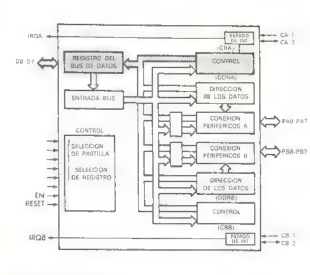
Figura 7.4
Empleo de un PIO: lectura del estado.

Figura 7.5
Empleo de un PIO: lectura de la entrada.



y o no un byte válido a la

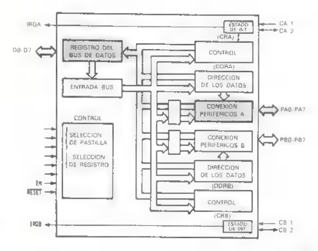
Figura 7.4
Empleo de un PIO: lectura del estado



Lectura de la puerta

El byte se lee en el Z80 (véase figura 7.5).

Figura 7.5
Empleo de un PIO: lectura de la entrada.



El PIO Zilog Z80

Es un dispositivo de dos puertas, de arquitectura esencialmente compatible con la que acabamos de describir como estándar. La disposición de las patillas aparece en la figura 7.6 y el diagrama de bloques en la 7.7.

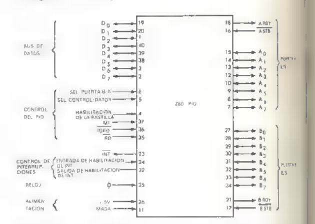


Figura 7.6
Patillas del PIO del Z80.

Cada una de las puertas del PIO tiene seis registros: uno de entrada, de 8 bits; otro de salida, también de 8 bits; uno de control en modo, de 2 bits; uno de máscara, de 8 bits; uno de selección entrada/salida (dirección), de 8 bits, y uno de control de la máscara, de 2 bits. Los tres últimos sólo se usan cuando la puerta está programada para funcionar en modo bit.

Las puertas tienen cuatro modos de funcionamiento, que se seleccionan mediante el registro de control en modo de 2 bits: salida de byte, entrada de byte, *bus* bidireccional y bit.

Los dos bits del registro de control de la máscara los carga el programador y especifican el estado alto o bajo del periférico que debe controlarse y las condiciones en que puede generarse una interrupción.

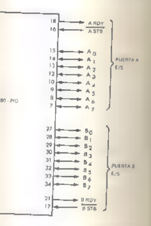
El registro selector de entrada/salida de 8 bits permite disponer cada una de las patillas en dirección de entrada o de salida cuando se opera en modo bit.

PROGRAMACION DEL PIO ZILOG

Una secuencia para usar el PIO en modo bit, por ejemplo, sería:

Figura 7.7
Diagrama de bloques del PIO Z80.

ertas, de arquitectura esencial-
acabamos de describir como
tillas aparece en la figura 7.6 y
7.



O tiene seis registros: uno de
también de 8 bits; uno de
máscara, de 8 bits; uno de
de ▮ bits, y uno de control
últimos sólo se usan cuando
uncionar en modo bit.
s de funcionamiento, que se
control en modo de 2 bits:
us bidireccional y bit.
rol de la máscara los carga
do alto o bajo del periférico
nes en que puede generarse

/salida de ▮ bits permite
dirección de entrada ▮ de
it.

LOG

en modo bit, por ejemplo,

Cargar el registro de control en modo para especificar el modo bit.

Cargar el registro selector de entrada/salida de la puerta A, para especificar que las líneas 0-5 son entradas y las 6 y 7 salidas.

A continuación podría leerse una palabra leyendo el contenido de la memoria auxiliar de entrada.

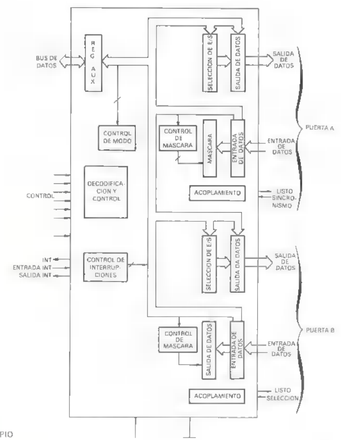


Figura 7.7
Diagrama de bloques del PIO Z80

Además, podría utilizarse el registro de filtro para especificar las condiciones de estado.

El lector interesado en una descripción detallada del funcionamiento del PIO deberá consultar el libro de esta misma colección *Z80 Applications Book.*

El SIO Z80

El SIO (entrada y salida en serie) es un periférico de dos canales construido para facilitar las comunicaciones asíncronas en serie. Contiene un receptor-transmisor asíncrono universal (UART), y su función principal es efectuar la conversión de serie a paralelo, y viceversa. No obstante, la pastilla dispone de recursos muy refinados, como la manipulación automática de protocolos complejos organizados por bytes, como IBM bisíncrono, o HDLC y SDLC, dos protocolos organizados por bits.

Además, puede funcionar en modo síncrono, como un USRT, y generar y verificar códigos CRC. Dispone de los modos muestreo, interrupción y transferencia de bloques. La descripción completa de este dispositivo está fuera del alcance de este texto introductorio, y aparece en el mencionado *Z80 Applications Book.*

OTRAS PASTILLAS DE E/S

Dado que el Z80 se utiliza habitualmente como sustituto del 8080, se ha construido de forma que pueda combinarse con casi todas las pastillas de E/S de éste, además de con las suyas específicas fabricadas por Zilog. Todas las pastillas de E/S 8080 pueden considerarse compatibles con un sistema Z80.

Resumen

Para sacar partido a los componentes de entrada/salida hay que conocer perfectamente la función de cada bit o grupo de bits de los diversos registros de control. Estas nuevas pastillas, de estructura compleja, automatizan muchas operaciones que antes se confiaban al soporte lógico o a alguna lógica especial. Más en concreto, los componentes como el SIO automatizan muchas de las operaciones de acoplamiento. Con la información de este capítulo, el lector estará en condiciones de entender las funciones de las señales y los registros básicos. Como es natural, seguirán apareciendo nuevos componentes que confiarán al soporte físico la ejecución de algoritmos todavía más complicados.

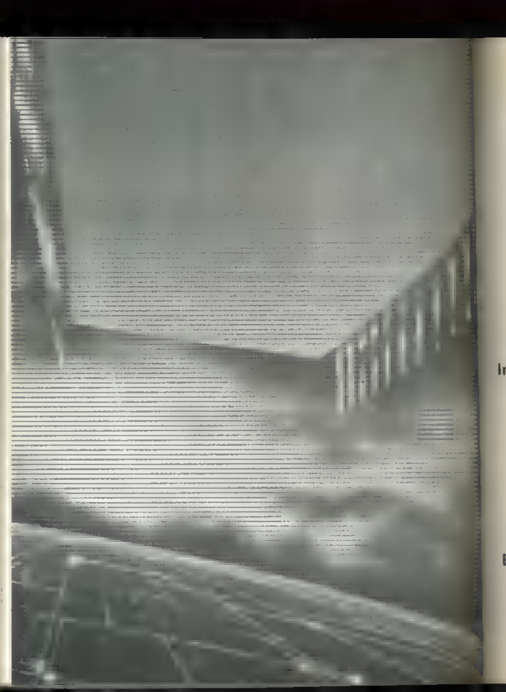
gistro de filtro para especificar

scripción detallada del funcio-
ltar el libro de esta misma

erie) es un periférico de dos
s comunicaciones asíncronas
nsmisor asíncrono universal
fectuar la conversión de serie
ante, la pastilla dispone de
manipulación automática de
por bytes, como IBM bisín-
ocolos organizados por bits.
modo síncrono, como un
igos CRC. Dispone de los
ransferencia de bloques. La
sitivo está fuera del alcance
rece en el mencionado *Z80*

ualmente como sustituto del
pueda combinarse con casi
además de con las suyas
las las pastillas de E/S 8080
on un sistema Z80.

entes de entrada/salida hay
n de cada bit o grupo de
trol. Estas nuevas pastillas,
n muchas operaciones que
o a alguna lógica especial.
como el SIO automatizan
lamiento. Con la informa-
en condiciones de entender
egistros básicos. Como es
componentes que confía-
a algoritmos todavía más



Introducción

Borrado de u

8 Aplicaciones

Introducción

Este capítulo presenta una serie de programas de aplicación práctica pensados para poner a prueba sus conocimientos de programación. Estos programas o "rutinas" se encuentran con frecuencia en aplicaciones más amplias y se conocen como "rutinas de servicio". Su creación obliga a una síntesis de los conocimientos y técnicas adquiridos en los anteriores capítulos.

Tendremos ocasión de tomar caracteres de dispositivos de E/S para someterlos a distintos tratamientos, pero antes de llegar a eso aprenderemos a borrar un área de la memoria (cosa que no siempre será necesaria; téngase en cuenta que estos programas se presentan exclusivamente como ejercicios de programación).

Borrado de una sección de memoria

Deseamos borrar —igualar a cero— los contenidos de la memoria comprendidos entre las direcciones BASE y BASE ± LONGITUD, siendo LONGITUD inferior a 256. El programa es:

```
CEROM   LD    B, LONGITUD   CARGAR LONGI-
                            TUD EN B
        LD    A, 0          BORRAR A
        LD    HL, BASE      APUNTADOR A
                            LA BASE
BORRAR  LD    (HL), A       BORRAR LA POSI-
                            CION A
        INC   HL            APUNTADOR A
                            LA POSICION SI-
                            GUIENTE
        DEC   B             DECREMENTAR
                            EL CONTADOR
        JR    NZ, BORRAR    ¿FIN DE LA SEC-
                            CION?
        RET
```

En el programa se supone que la longitud de la sección de memoria es igual a LONGITUD. El par de registros HL se utiliza como apuntador ■ la palabra en curso que debe borrarse. El registro B, como es habitual, se emplea como contador.

El acumulador A se carga una sola vez con el valor 0 (todo ceros), y a continuación se copia en posiciones de memoria sucesivas.

En un programa de verificación de memoria, por ejemplo, esta rutina de servicio serviría para poner ■ 0 el contenido de un bloque; a continuación, el programa comprobaría, de la forma habitual, ■ su contenido sigue siendo 0.

Lo que acabamos de ver es la ejecución normal de una rutina de borrado, que todavía puede mejorarse, como demuestra esta nueva versión:

```
CEROM   LD     B, LONGITUD
        LD     HL, BASE
BUCLE   LD     (HL), 0
        INC    HL
        DJNZ   BUCLE
        RET
```

Se han introducido dos mejoras, que consisten en la eliminación de la instrucción LD A, 0, en la carga de un "0" directamente en la posición señalada por H y L y en el uso de la instrucción especial del Z80 DJNZ.

Este ejemplo pone de relieve que *la primera versión de un programa, aunque sea perfectamente correcta, casi siempre puede mejorarse estudiándola atentamente*. Para introducir mejoras es imprescindible estar muy familiarizado con todas las instruccio-

Muestreo de

nes. Es importante señalar que las mejoras no son sólo cosméticas, sino que realmente aumentan la velocidad de funcionamiento del programa, reducen el número de instrucciones y, por tanto, el espacio de memoria, y, por lo general, aumentan de paso la legibilidad; por consiguiente, la facilidad de corrección del mismo.

Ejercicio 8.1: *Escriba un programa de verificación de memoria que iguale a 0 un bloque de 256 palabras y que a continuación compruebe si hay un 0 en cada posición. Acto seguido escribirá todo unos, y volverá a comprobar el contenido del bloque. Hecho esto repetirá el proceso escribiendo 01010101 y 10101010.*

Ejercicio 8.2: *Modifíquese el programa del ejercicio anterior en el sentido de ocupar la sección de memoria primero todo con ceros y a continuación todo con unos.*

Vamos ahora a hacer un muestreo de los dispositivos de E/S para averiguar cuáles necesitan servicio.

Muestreo de dispositivos de E/S

Supondremos que tales dispositivos están conectados a nuestro sistema. Sus registros de estado se encuentran en las direcciones ESTADO1, ESTADO2 y ESTADO3. El programa es el siguiente:

PRUEBA	IN	A,(ESTADO1)	LEER ESTADO1 E/S
	BIT	7,A	TEST BIT "LISTO" (BIT 7)
	JP	NZ,UNO	SALTAR A MANIPULADOR 1
	IN	A,(ESTADO2)	LO MISMO PARA DISPOSITIVO 2
	BIT	7,A	
	JP	NZ,DOS	
	IN	A,(ESTADO3)	LO MISMO PARA EL DISPOSITIVO 3
	BIT	7,A	
	JP	NZ,TRES	
	(salida sin resultado)		

Como resultado de la instrucción BIT, el bit Z de las banderas de estado se activa a 1 si el ESTADO ■ 0. JP NZ (salto si no es igual a 0) provoca una bifurcación a la rutina ENE correspondiente.

Introducción de caracteres

Supongamos que acabamos de observar que hay un carácter listo en el teclado; vamos a acumular varios más en un área de memoria llamada DEPOSITO hasta que encontremos uno especial, SPC, cuyo código ya ha sido definido.

La subrutina TOMCAR toma un carácter del teclado (véase el capítulo 6 para más detalles) y lo deposita en el acumulador. Supongamos que no pueden tomarse más de 256 caracteres sin que aparezca SPC.

```
SERIE    LD     HL, DEPOSITO   SEÑALA AL DEPOSITO
BUCLE    CALL   TOMCAR         TOMAR UN CARACTER
         CP     SPC            VERIFICAR ■ ES EL ESPECIAL JR
         JR     Z, FUERA       ¿HALLADO?
         LD     (HL), A        ALMACENAR CAR EN EL DEPOSITO
         INC    HL             SIGUIENTE POS DEPOSITO
         JR     BUCLE          TOMAR SIGUIENTE CAR
FUERA    RET
```

Ejercicio 8.3: *Tratemos de mejorar esta rutina básica:*

a) *Devuélvase el carácter al dispositivo (un teletipo, por ejemplo).*

b) *Compruébese que la serie de entrada no tiene más de 256 caracteres.*

Ya tenemos una serie de caracteres en la memoria auxiliar (DEPOSITO), que podemos someter ■ diversos tratamientos.

Verificación de un carácter

Vamos a determinar si el carácter situado en la posición de memoria LOC es igual ■ 0, ■ 1 o a 2:

```
CUD     LD      A,(LOC)         TOMAR UN CARACTER
        CP      00              ¿ES CERO?
        JP      Z, CERO         SALTAR A RUTINA
        CP      01              ¿ES UNO?
        JP      Z, UNO
        CP      02              ¿ES DOS?
        JP      Z, DOS
        JP      NOVISTO         FALLO
```

Nos hemos limitado a leer el carácter y a aplicar la instrucción CP para comprobar su valor.

Vamos a realizar ahora una berificación de diferente naturaleza.

Verificación de intervalo

Se trata de determinar si el carácter ASCII que ocupa la posición de memoria LOC es una cifra comprendida entre 0 y 9:

```
INTER   LD      A,(LOC)         TOMAR UN CARAC-
                                TER
        AND     7FH             FILTRAR EL BIT DE
                                PARIDAD
        CP      30H             ASCII 0
        JR      C, FUERA        ¿CARACTER MUY
                                BAJO?
        CP      39H             ASCII 9
        JR      NC, FUERA       ¿CARACTER MUY
                                ALTO?
        CP      A               FORZAR BANDERA 0
FUERA   RET     SALIDA
```

El carácter ASCII "0" se representa en hexadecimal por "30" o por "B0", según se use o no bit de paridad. De la misma forma, el carácter ASCII "9" se representa como "39" o como "B9".

El objetivo de la segunda instrucción del programa es borrar el bit de paridad 7, si es que se estaba utilizando, para que el programa sea aplicable en cualquier caso. A continuación se compara el valor del carácter con los ASCII "0" y "9"; al emplear una instrucción de comparación, la bandera Z se activa a 1 si el resultado es positivo. El bit de acarreo se activa si hay acarreo, y se quita en caso contrario. En otras palabras: cuando se usa una instrucción CP, el bit de acarreo se activa ■ el valor

del literal que aparece en la instrucción es superior al valor contenido en el acumulador, y se pone a "0" si es igual o menor.

La última instrucción, CP A, fuerza un "1" en la bandera Z, que se usa para indicar a la rutina de llamada que el carácter de CAR está realmente comprendido en el intervalo 0-9. Puede emplearse cualquier otra convención, como cargar una cifra en el acumulador para indicar el resultado de la comprobación.

Ejercicio 8.4: *¿Equivale al de arriba el siguiente programa?:*

```
LD      A,(CAR)
SUB     30H
JP      M,FUERA
SUB     10D
JP      P,FUERA
ADD     10D
```

Ejercicio 8.5: *Determínese si un carácter ASCII contenido en el acumulador es o no una letra del alfabeto.*

Observará que en las tablas ASCII se emplea con frecuencia la paridad; así, el equivalente del ASCII "0" es "0110000", un código de 7 cifras. Pero trabajando en paridad impar, por ejemplo, se garantiza que el número total de unos de una palabra es impar; el código pasaría a ser "10110000" (se ha incorporado un "1" adicional a la izquierda), que equivale a "B0" en hexadecimal. Vamos, pues, a crear un programa generador de paridad.

Generación de paridad

Este programa genera paridad par con el bit 7:

```
PARIDAD   LD     A,(CAR)     TOMAR UN CA-
                             RACTER
          AND    7FH         BORRAR EL BIT
                             DE PARIDAD
          JP     PE,FUERA    VERIFICAR SI LA
                             PARIDAD YA ES
                             PAR
          OR     80H         ACTIVAR EL BIT
                             DE PARIDAD
FUERA     LD     (LOC),A     ALMACENAR EL
                             RESULTADO
```


Conversión d

El programa utiliza el circuito interno de detección de paridad de que dispone el Z80.

La tercera instrucción —JP PE, OUT— comprueba si la paridad de la palabra del acumulador es ya par o no; el resultado es positivo si la respuesta es afirmativa (PE), lo que da lugar a la salida (FUERA).

Si la paridad no es par, es decir, si la instrucción de salto no se ejecuta, significa que es impar y que debe escribirse un "1" en el bit 7, operación de la que se encarga la cuarta instrucción:

```
OR    80H
```

Por último, el valor resultante se guarda en la posición de memoria LOC.

Ejercicio 8.6: *Con el circuito interno de detección de paridad, el problema anterior resulta demasiado sencillo de resolver. A modo de ejercicio, trate ahora de solucionarlo sin contar con dicho circuito; desplace el contenido del acumulador y cuente el número de unos para determinar el bit que debe escribirse en la posición de paridad.*

Ejercicio 8.7: *Con el programa anterior como ejemplo, verifique la paridad de una palabra. Debe calcular la paridad correcta y compararla con la esperada.*

Conversión de código: ASCII a BCD

Pasar el código ASCII al BCD es muy sencillo. Observará que la representación hexadecimal de los caracteres ASCII comprendidos entre 0 y 9 va de 30 a 39 o de B0 a B9, dependiendo de la paridad. La representación BCD se obtiene simplemente añadiendo el "3" o la "B", es decir, filtrando el *nibble* (cuatro bits) de la izquierda:

ASCBCD	CALL	INTER	COMPROBAR SI EL CAR ESTA ENTRE 0 Y 9
	JP	NZ, ILEGAL	SALIR SI CAR ES ILEGAL
	AND	0FH	FILTRAR *NIBBLE* SUPERIOR
	LD	(BCDCAR), A	ALMACENAR EL RESULTADO

Ejercicio 8.8: Escríbase un programa para pasar de BCD a ASCII.

Ejercicio 8.9: Escríbase un programa para pasar de BCD a binario (es un problema más difícil de resolver).

Un consejo: el BCD $N_3 N_2 N_1 N_0$ equivale al binario $(((N_3 \times {}^{10}) + N_2) \times 10 + N_1) \times 10 + N_0$.

Para multiplicar por 10 se hace un desplazamiento a la izquierda (= ×2), otro más (= ×4), una instrucción ADC (= ×5) y otro desplazamiento a la izquierda (= ×10).

En notación BCD completa, la primera palabra puede contener la cuenta de cifras BCD, el *nibble* siguiente el signo y cada uno de los demás *nibbles* una cifra BCD (suponemos que no hay coma decimal). El último *nibble* del bloque puede permanecer sin utilizar.

Figura 8.1
El mayor elemento de una tabla.

Conversión de hexadecimal a ASCII

"A" contiene una cifra hexadecimal, y no tenemos más que añadir un "3" (o una "B") al *nibble* izquierdo:

```
AND   0FH        CERO EN EL NIBBLE IZ-
                 QUIERDO (OPCIONAL)
ADD   A,30H      ASCII
CP    3AH        ¿NECESARIA CORRECCION?
JP    M,FUERA
ADD   A,7        CORRECCION DE A A F
```

Ejercicio 8.10: Realícese la conversión de HEX a ASCII suponiendo que se trabaja en formato empaquetado (dos cifras hexadecimales en A).

Búsqueda del elemento mayor de una tabla

La dirección de partida de la tabla se encuentra en la dirección de memoria BASE. La primera entrada de la misma es el número de bytes que contiene. El programa se encargará de buscar el mayor elemento de la misma, del que depositará el valor en A y la posición en la dirección de memoria INDICE.

El programa utiliza los registros A, F, B, H y L, y utiliza direccionamiento indirecto para poder buscar la tabla en cualquier lugar de la memoria (véase figura 8.1).

Figura 8.1
El mayor elemento de una tabla

```
MAX     LD    HL, BASE       DIRECCION DE LA
                             TABLA
        LD    B,(HL)         NUMERO DE BY-
                             TES DE LA TABLA
        LD    A,0            BORRAR VALOR
                             MAXIMO
        INC   HL             INICIALIZAR INDI-
                             CE
        LD    (INDICE),HL    ENTRADA SI-
                             GUIENTE
BUCLE   CP    (HL)           COMPARAR EN-
                             TRADA
        JR    NC,CAMBIO      SALTA SI ES ME-
                             NOR QUE EL MA-
                             XIMO
        LD    A,(HL)         CARGA NUEVO
                             VALOR MAXIMO
        LD    (INDICE),HL    CARGA NUEVO
                             VALOR MAXIMO
CAMBIO  INC   HL             SEÑALA LA EN-
                             TRADA SIGUIENTE
        DEC   B              DECREMENTA EL
                             CONTADOR
        JR    NZ,BUCLE       SIGUE SI NO ES 0
        RET
```

El programa comprueba la entrada enésima: si es mayor que 0, pasa a A, y su posición se recuerda en INDICE; a continuación se comprueba la entrada n - 1, y así sucesivamente. El programa funciona con enteros positivos.

Ejercicio 8.11: *Modifíquese el programa para que funcione también con números negativos en complemento a dos.*

Ejercicio 8.12: *¿Funcionará el programa con caracteres ASCII?*

Ejercicio 8.13: *Escríbase un programa que clasifique n números en orden ascendente.*

Ejercicio 8.14: *Escríbase un programa que clasifique n nombres de 3 caracteres cada uno en orden alfabético.*

Suma de N elementos

El programa que veremos a continuación calcula la suma de 16 bits de N entradas positivas de una tabla. La dirección de partida de la misma se encuentra en BASE y su primera entrada contiene el número de elementos N. La suma de 16 bits se deposita en las posiciones de memoria SUMBJ y SUMAL. Si requiriese más de 16 bits, sólo se conservarían los 16 inferiores (en tal caso se dice que los superiores se truncan).

El programa modificará los registros A, F, B, H, L, IX, y supone que el número máximo de elementos no pasa de 256 (véase figura 8.2).

```
SUMN       LD    HL,BASE       APUNTA LA
                               BASE DE LA
                               TABLA
           LD    B,(HL)        LEE LA LON-
                               GITUD EN EL
                               CONTADOR
SUMIG      INC   HL            APUNTA A
                               LA PRIMERA
                               ENTRADA
           LD    IX,SUMBJ      APUNTA AL
                               RESULTADO,
                               INF
           LD    (IX + 0),0    BORRA RE-
                               SULTADO,
                               INFERIOR
           LD    (IX + 1),0    Y SUPERIOR
BUCLESUM   LD    A,(HL)        TOMA EN-
                               TRADA DE
                               LA TABLA
           ADD   A,(IX + 0)    CALCULA
                               SUMA PAR-
                               CIAL
```

Figura 8.2
Suma de N elementos.

```
            LD     (IX + 0), A       LA ALMACE-
                                     NA EN OTRO
                                     SITIO
            JR     NC, NOACARR       COMPRUEBA
                                     SI HAY ACA-
                                     RREO
            INC    (IX + 1)          SUMA ACA-
                                     RREO
                                     AL BYTE SUP
NOACARR     INC    HL                APUNTA A
                                     LA SIGUIEN-
                                     TE ENTRADA
            DEC    B                 DECREMEN-
                                     TA EL CON-
                                     TADOR DE
                                     BYTES
            JR     NZ, BUCLESUM      SIGUE SU-
                                     MANDO
                                     HASTA EL FI-
                                     NAL
            RET
```

Se trata de un programa muy sencillo que no precisa de más explicaciones.

Ejercicio 8.15: *Modifíquese el programa anterior para:*

a) *Calcular sumas de 24 bits.*
b) *Calcular sumas de 32 bits.*
c) *Detectar cualquier desbordamiento.*

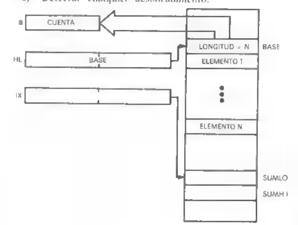


Figura 8.2
Suma de N elementos.

Cálculo del total de control

El total de control es una cifra o un grupo de cifras que se calculan a partir de un bloque de caracteres sucesivos. Dicho total se va calculando conforme se van almacenando los datos, y se coloca al final de los mismos. Para comprobar la integridad de un dato, se lee, se vuelve a calcular el total de control y se compara con el valor almacenado; si hay discrepancia, [illegible] que se ha producido un error o un fallo.

Para calcularlo se emplean varios algoritmos. En este caso aplicaremos la operación OR exclusivo a todos los bytes de una tabla de N elementos y dejaremos el resultado en el acumulador. Como es habitual, el inicio de la tabla reside en la dirección BASE, y la primera entrada de la misma es el número de elementos N. El programa modifica los registros A, F, B, H, L; N debe ser inferior a 256.

```
TCONT   LD    HL, BASE       CARGA LA DIREC-
                             CION DE LA TABLA
                             EN HL
        LD    B,(HL)         HACE  N = LONGI-
                             TUD
        XOR   A              BORRA EL TOTAL
                             DE CONTROL
        INC   HL             APUNTA AL PRI-
                             MER ELMENTO
BUCLE   XOR   (HL)           CALCULA EL TO-
                             TAL DE CONTROL
        INC   HL             APUNTA AL SI-
                             GUIENTE ELEMEN-
                             TO
        DEC   B              DECREMENTA EL
                             CONTADOR
        JR    NZ, BUCLE      REPITE SI NO HA
                             TERMINADO
        LD    (TCONT), A     CONSERVA EL TO-
                             TAL DE CONTROL
        RET
```

Cómputo de ceros

El programa cuenta el número de ceros de nuestra tabla habitual y lo deposita en la posición TOTAL. Modifica A, B, C, H, L, F.

```
CEROS     LD    HL, BASE      APUNTA A LA
                              TABLA
          LD    B,(HL)        LEE LA LONGI-
                              TUD EN EL
                              CONTADOR
          LD    C,0           TOTAL A 0
          INC   HL            APUNTA A LA
                              PRIMERA EN-
                              TRADA
BUCCERO   LD    A,(HL)        TOMA UN ELE-
                              MENTO
          OR    0             ACTIVA BAN-
                              DERA 0
          JR    NZ, NOCERO    ¿ES 0?
          INC   C             SI LO ES, IN-
                              CREMENTA EL
                              CONTADOR DE
                              CEROS
NOCERO    INC   HL            APUNTA A LA
                              SIGUIENTE EN-
                              TRADA
          DEC   B             DECREMENTA
                              EL CONTADOR
                              DE LONGITUD
          JR    NZ, BUCCERO
          LD    A,C
          LD    (TOTAL),A     ARCHIVARLO
```

Ejercicio 8.16: *Modifíquese el programa para contar:*

a) *El número de asteriscos (carácter "*").*
b) *El número de letras del alfabeto.*
c) *El número de cifras que hay entre "0" y "9".*

Transferencia de bloques

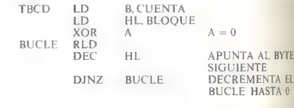
El programa coge una entrada de cada tres del bloque fuente situado en la dirección DESDE y las almacena en bloque en la dirección HASTA:

```
DCADA3    LD    HL, DESDE
          LD    DE, HASTA     ACTIVA LOS
                              APUNTADORES
          LD    BC, TAMAÑO
BUCLE     LDI                 TRANSFEREN-
                              CIA AUTOMATI-
                              CA
```

```
INC   HL
INC   HL              SALTA 2 ENTRA-
                      DAS
JP    PE, BUCLE
```

Transferencia de bloques en BCD

Se trata de introducir en la memoria varias cifras BCD, es decir, de desplazar varios *nibbles* (véase figura 8.3). El programa aparece a continuación:



```
TBCD    LD      B, CUENTA
        LD      HL, BLOQUE
        XOR     A               A = 0
BUCLE   RLD
        DEC     HL              APUNTA AL BYTE
                                SIGUIENTE
        DJNZ    BUCLE           DECREMENTA EL
                                BUCLE HASTA 0
```

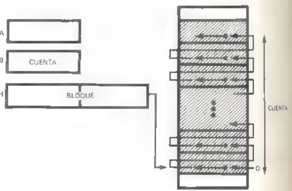


Figura 8.3
Transferencia de bloques en BCD: memoria.

El programa utiliza la instrucción RLD, que todavía no habíamos usado. RLD produce una rotación a la izquierda de la cifra BCD entre A y (HL). (HL) o M denotan los contenidos de las posiciones de memoria apuntadas por H y L.

MINF va a MSUP.
MSUP va a AINF.
AINF va a MINF.

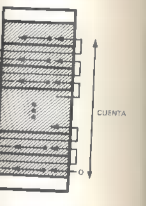
SALTA 2 ENTRA-
DAS

.E

noria varias cifras BCD, es
ase figura 8.3). El programa

A
UE
A = 0

APUNTA AL BYTE
SIGUIENTE
DECREMENTA EL
BUCLE HASTA 0



RLD, que todavía no ha-
ción a la izquierda de la
enotan los contenidos de
por H y L.

En este caso, "inf" y "sup" hacen referencia a *nibbles* de 4 bits.

Para utilizar la potente instrucción DJNZ, se ha empleado el registro B como contador de cifras. HL se activa para que apunte al principio del bloque.

A se emplea para almacenar la cifra izquierda de desplazamiento en cada rotación entre dos accesos sucesivos al bloque.

Por convenio, "0" se introduce en el bloque por la parte inferior.

Comparación de dos números de 16 bits con signo

IX apunta al primer número N1.

IY apunta a N2 (véase figura 8.4).

El programa activa el bit de arrastre si N1 < N2, y el bit Z si N1 = N2.

```
COMP    LD    B,(IX + 1)    TOMA EL SIGNO
                            DE N1
        LD    A, B
        AND   80H           VERIFICA EL SIG-
                            NO, BORRA CY
        JR    NZ, NEGM1     INVIERTE N1
        BIT   7,(IY + 1)
        RET   NZ            INVIERTE N2
        LD    A, B
        CP    (IY + 1)      AMBOS SIGNOS
                            POSITIVOS
        RET   NZ
        LD    A,(IX)
        CP    (IY)
        RET
NEGM1   XOR   (IY + 1)
        RLA                 BIT DE SIGNO A
                            CY
        RET   C             SIGNOS DIFE-
                            RENTES
        LD    A, B
        CP    (IY + 1)      AMBOS SIGNOS
                            NEGATIVOS
        RET   NZ
        LD    A,(IX)
        CP    (IY)
        RET
```

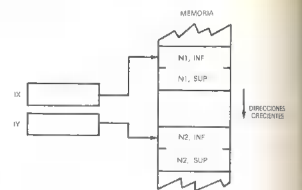


Figura 8.4
Comparación de dos números con signo.

El programa empieza por comprobar los signos de N1 y N2. Si N1 es negativo, se da un salto a NEGM1: en caso contrario, se ejecuta la parte superior del programa.

Obsérvese que en la quinta línea se utiliza la instrucción BIT para verificar directamente el bit del signo de N2 en la memoria:

```
BIT    7,(IY + 1)
```

Podría haberse hecho lo mismo con N1, pero como el valor de éste hace falta rápidamente, es más sencillo leerlo en memoria y conservarlo en B:

```
COMP    LD    B,(IX + 1)
```

Es necesario conservar N1 en B porque AND puede destruir el contenido de A:

```
LD      A, B
AND     80H
```

Obsérvese que se utiliza un retorno condicional en la línea 6:

```
RET    NZ
```

Es un recurso poderoso del Z80 que simplifica la programación.



probar los signos de N1 y
salto a NEGM1; en caso
rior del programa.
ea se utiliza la instrucción
bit del signo de N2 en la

on N1, pero como el valor
ás sencillo leerlo en memo-

orque AND puede destruir

rno condicional en la lí-

■ simplifica la programa-

Obsérvese que la instrucción de comparación actúa directamente sobre el contenido de la memoria en modo indexado:

CP (IY + 1)

Al comparar los dos números, se empieza por el byte más significativo y se pasa a continuación al menos.

Obsérvese el generoso empleo del mecanismo de indexación que se hace en el programa y que da lugar a una codificación muy eficaz.

Ordenación por burbuja

Es una técnica de ordenación que sirve para organizar los elementos de una tabla en orden ascendente o descendente. Debe su nombre a que los elementos menores "flotan" por entre los demás hasta la parte superior de la tabla. Cada vez que un elemento menor "choca" con otro "más pesado", salta por encima del mismo.

La figura 8.5 recoge un ejemplo práctico de burbuja. La lista que debe ordenarse contiene los elementos (10, 5, 0, 2, 100) y debe organizarse en orden descendente (de forma que el "0" quede en la parte superior). El algoritmo es sencillo y el diagrama de flujo aparece en la figura 8.7.

Se comparan los dos elementos superiores (o los dos inferiores); si el inferior es menor ("más ligero") que el superior, se intercambian; en caso contrario, se dejan como están. Por razones prácticas, el intercambio, si se produce, se recuerda en una bandera llamada "CAMBIADO". La operación se repite con los dos elementos siguientes, a continuación con los otros dos y así sucesivamente, hasta haber comparado todos dos a dos.

El primer paso se ilustra en las fases 1, 2, 3, 4, 5 y 6 de la figura 8.5; la operación avanza de abajo arriba, pero igualmente podría haberse hecho al revés.

Si no se produce ningún intercambio, es que la ordenación está terminada. Si se produce alguno, hay que empezar de nuevo con las comparaciones.

En la figura 8.6 se observa que el ejemplo propuesto requiere cuatro pasadas. Se trata de un proceso sencillo y muy usado.

Hay una complicación adicional debida al propio mecanismo de intercambio. En efecto, al intercambiar A y B no puede escribirse

A = B
B = A

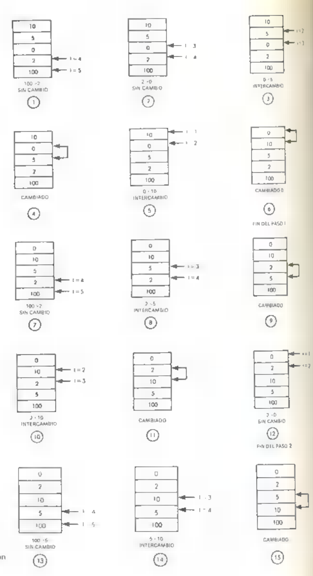
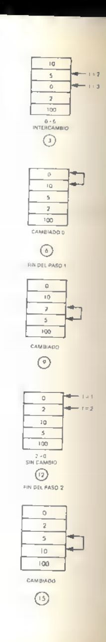


Figura 8.5
Fases 1 a 12 de la ordenación por burbuja.

Figura 8.6
Fases 16 a 21 de la ordenación por burbuja.



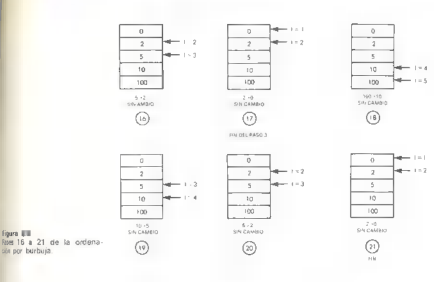


Figura
Fases 16 a 21 de la ordenación por burbuja.

porque ello daría lugar a la pérdida del valor anterior de A (pruebe a hacerlo con un ejemplo).

La solución correcta es utilizar una variable o una posición provisionales para conservar el valor de A:

PROV = A
A = B
B = PROV

Pruebe con un ejemplo, y verá que funciona; esta técnica se llama permutación circular.

Todos los programas realizan el intercambio de esta forma, que ilustra el diagrama de flujo de la figura 8.7.

La distribución de registros aparece en la figura 8.8: el programa es el siguiente:

```
BURBUJA   LD    (PROV), HL      PROV = (HL)
OTRAVEZ   LD    IX, (PROV)      IX = (HL)
          RES   CAMBIO, H       INTERCAMBIO
                                BANDERA = 0
          LD    B, C
          DEC   B
```

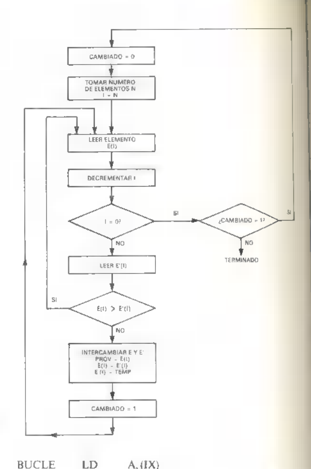
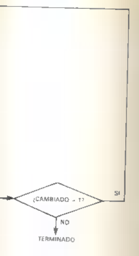


Figura 8.7
Diagrama de flujo de la ordenación por burbuja.

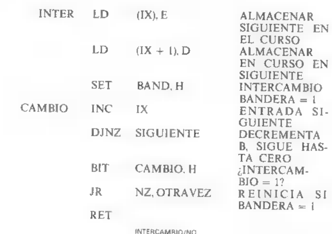
```
BUCLE   LD   A,(IX)
        LD   D,A           D = ENTRADA
                           EN CURSO
        LD   E,(IX + 1)    E = ENTRADA
                           SIGUIENTE
        CP   E             COMPARA-
                           CION
        JR   NC,CAMBIO     IR A CAMBIO SI
                           EN CURSO ≥
                           SIGUIENTE
```

Figura 8.8
Ordenación por burbuja.

Resumen



D = ENTRADA EN CURSO
E = ENTRADA SIGUIENTE
COMPARACION
IR A CAMBIO SI EN CURSO ≥ SIGUIENTE



```
         INTER   LD     (IX), E        ALMACENAR SIGUIENTE EN EL CURSO
                 LD     (IX + 1), D    ALMACENAR EN CURSO EN SIGUIENTE
                 SET    BAND, H        INTERCAMBIO BANDERA = 1
         CAMBIO  INC    IX             ENTRADA SIGUIENTE
                 DJNZ   SIGUIENTE      DECREMENTA B, SIGUE HASTA CERO
                 BIT    CAMBIO, H      ¿INTERCAMBIO = 1?
                 JR     NZ, OTRAVEZ    REINICIA SI BANDERA = 1
                 RET
```

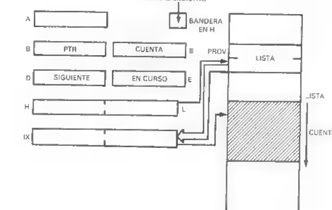
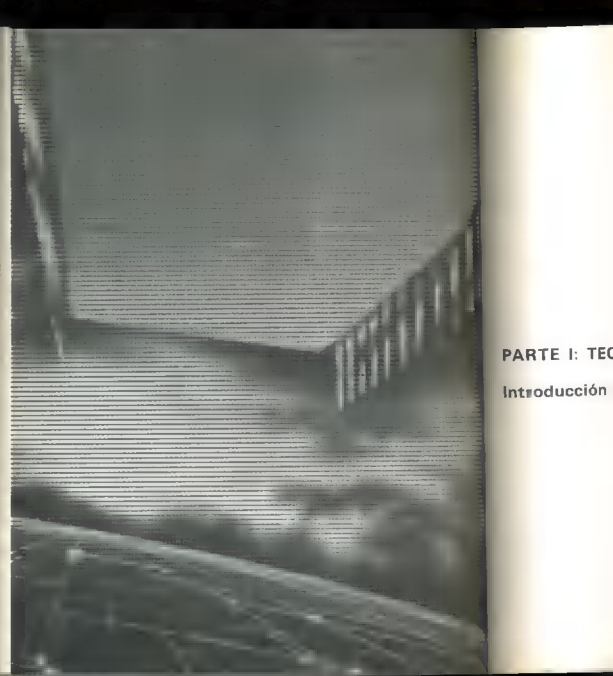


Figura
Ordenación por burbuja.

Resumen

Hemos visto en este capítulo una serie de rutinas de servicio que combinan técnicas ya estudiadas en otros anteriores y que deben permitir al lector empezar a crear sus propios programas. Muchas de las rutinas utilizan una estructura de datos llamada tabla, que, como veremos en el capítulo siguiente, no es la única que existe.



PARTE I: TEC

Int▪oducción

9

Estructuras de datos

PARTE I: TEORIA

Introducción

El diseño de un buen programa supone dos tareas: *diseño del algoritmo y diseño de las estructuras de datos*. En los más sencillos no existen estructuras de datos de importancia, por lo que la tarea fundamental se reduce a diseñar el algoritmo y codificarlo acertadamente en un lenguaje máquina determinado, que es lo que hemos hecho hasta el momento. Sin embargo, para hacer programas de cierta complejidad es necesario conocer las estructuras de datos. Dos de ellas ya las hemos usado frecuentemente: la tabla y la pila. La finalidad de este capítulo es presentar otras más generales que puedan resultar de utilidad. El texto del mismo es completamente independiente del microprocesador y del ordenador que se utilice, puesto que su contenido es teórico y se refiere a la organización lógica de los datos en el sistema. Hay libros dedicados exclusivamente al estudio de las estructuras de datos, de la misma forma que los hay especializados en eficacia de computación o algorítmos de división y otras operaciones habituales; por tanto, nos limitaremos aquí a lo fundamental, sin pretender ser exhaustivos.

Punteros

Un puntero es un número que designa la posición de un dato. Todo puntero es una dirección, pero no necesariamente toda dirección es un puntero. Para serlo, debe señalar un dato o una información estructurada. Ya hemos trabajado frecuentemente con uno: el puntero de la pila, que señala la parte superior de la misma (o la posición situada inmediatamente por encima de dicha parte superior). Como veremos en seguida, la pila es una estructura de datos muy corriente llamada LIFO (*last in, first out:* último en entrar, primero en salir).

Cuando se trabaja con direccionamiento indirecto, la dirección indirecta es siempre un puntero que señala el dato que desea recuperarse.

Ejercicio 9.1: *Estudie la figura 9.1; en la dirección 15 de memoria hay un puntero que señala la tabla T, que empieza en la dirección 500. ¿Cuál es el verdadero contenido del puntero que señala hacia T?*

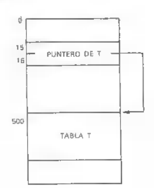


Figura 9.1
Puntero de dirección.

Listas

Casi todas las estructuras de datos están organizadas en forma de listas de diferente naturaleza.

LISTAS SECUENCIALES

La lista secuencial, también llamada tabla o bloque, es, probablemente, la estructura de datos más sencilla (ya utilizada en capítulos anteriores). Una tabla suele ordenarse en función

Figura 9.2
Estructura de un directorio.

de un criterio específico, como el orden alfabético o numérico, que facilita la labor de recuperar un elemento de la misma (mediante direccionamiento indexado, por ejemplo). Por bloque suele entenderse un grupo de datos de límites definidos, pero de contenido no ordenado; puede contener series de caracteres o ser un sector de un disco o un área lógica de memoria (llamada también segmento). En tales casos, no suele ser fácil acceder a un elemento aleatorio del bloque. Para facilitar la recuperación de bloques de información se utilizan directorios.

DIRECTORIOS

Un directorio es una lista de tablas o bloques; los archivos, por ejemplo, suelen seguir una estructura de directorio. A modo de ilustración, el directorio maestro de un sistema podría contener una lista de nombres de usuarios, como la ilustrada en la figura 9.2: la entrada del usuario "Juan" señala el directorio del archivo Juan, que consiste en una tabla que contiene los nombres de todos los archivos de Juan y sus posiciones; se trata, pues, de una tabla de punteros. En este caso, el directorio es de dos niveles, pero un sistema flexible debe permitir la inclusión de todos los directorios intermedios que puedan resultar cómodos para el usuario.

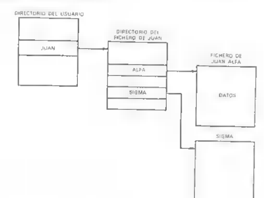


Figura 9.2
Estructura de un directorio.

LISTA ENCADENADA

En un sistema suele haber bloques de información que representan datos, acontecimientos u otras estructuras que no resultan fáciles de desplazar, pero que casi siempre pueden

reunirse en una tabla para clasificarlos o estructurarlos. El problema que se plantea es que nos interesa dejar todos los datos donde están y, a la vez, introducir un orden entre ellos (primero, segundo, tercero, etc.). La solución [redacted] este problema podría ser una lista encadenada, como la que se ilustra [redacted] la figura 9.3. Un puntero de la lista, llamado PRIMERBLOQUE, señala el primer bloque de la misma; dentro de este bloque 1 hay una posición reservada —la primera o la última palabra, por ejemplo— que contiene un puntero orientado hacia el bloque 2 y llamado PTR1; el mismo proceso se repite para los bloques 2 y 3. Dado que en el ejemplo que nos ocupa éste [redacted] el último de la lista, PTR3 contendrá, por convenio, o un valor especial "nulo" o un puntero orientado hacia sí mismo que permita detectar el final de la lista. Se trata de una estructura económica que sólo exige un puntero por bloque y ahorra al usuario la necesidad de tener que desplazar los bloques en la memoria.



Figura 9.3
Lista encadenada.

Estudiemos, por ejemplo, la inserción de un nuevo bloque, que muestra la figura 9.4. Supongamos que el tal bloque nuevo se encuentra en la dirección NUEBLOC y debe insertarse entre el bloque 1 y el bloque 2; basta adjudicar al puntero PTR1 el valor NUEBLOC para que señale hacia el bloque X; PTRX albergará el anterior valor de PTR1 y, por tanto, seguirá señalando al bloque 2. Los demás punteros de la estructura permanecen invariables, de modo que para insertar un bloque nuevo basta con actualizar dos punteros de la estructura.

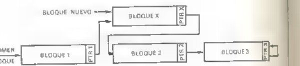


Figura 9.4
Inserción de un bloque nuevo.

Ejercicio 9.2: *Dibújese un diagrama que represente la extracción del bloque 2 de la estructura descrita.*

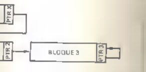
Se han desarrollado varios tipos de listas que facilitan formas de acceso, inserción y eliminación específicas, de las que a continuación examinaremos las más comunes.

Figura 9.5
Una cola.

carlos o estructurarlos. El
os interesa dejar todos los
ducir un orden entre ellos
. solución ■ este problema
mo la que se ilustra en la
, llamado PRIMERBLO-
la misma: dentro de este
a —la primera o la última
ne un puntero orientado
el mismo proceso se repite
el ejemplo que nos ocupa
contendrá, por convenio, o
ntero orientado hacia sí
de la lista. Se trata de una
un puntero por bloque y
ner que desplazar los blo-



ción de un nuevo bloque,
os que el tal bloque nuevo
OC y debe insertarse entre
dicar al puntero PTR1 el
hacia el bloque X; PTRX
y, por tanto, seguirá seña-
unteros de la estructura
e para insertar un bloque
teros de la estructura.



*ue represente la extracción
rita.*

de listas que facilitan for-
n específicas, de las que a
comunes.

COLA

La cola se denomina formalmente FIFO (*first in, first out:* primero en entrar, primero en salir), y su estructura se ilustra en la figura 9.5. Supongamos, para clarificar el diagrama, que el bloque de la izquierda es una rutina de servicio de un dispositivo de salida (una impresora, por ejemplo). Los bloques de la derecha permiten acceder a diferentes programas o rutinas de impresión; el orden en que se les atiende viene determinado por la cola de espera. Es fácil comprobar que quien primero obtendrá servicio será el bloque 1; después, el 2, y a continuación, el 3. El convenio establece que en una cola el último acontecimiento en llegar se sitúa al final de la misma (detrás de PTR3 en este caso). De esta forma se garantiza que el primero que se insertó en la misma será el primero en ser atendido. En sistemas informáticos es normal organizar en colas los acontecimientos que pueden esperar la atención de recursos escasos, como el procesador o algunos dispositivos de entrada/salida.

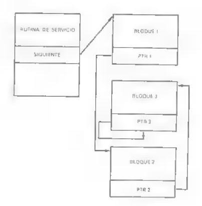


Figura 9.5
Una cola.

PILA

Esta estructura ya la hemos examinado detalladamente a lo largo de todo el libro: se denomina LIFO (último en entrar, primero en salir), porque el último elemento que se deposita en la parte superior de la misma es el primero que se extrae. La pila puede materializarse como bloque clasificado o como lista. Dado que en los microprocesadores la lista se emplea sobre

todo en acontecimientos de alta velocidad, como subrutinas o interrupciones, lo que suele alojarse en la misma no es una lista encadenada, sino un bloque continuo.

LISTA ENCADENADA FRENTE A BLOQUE

Una cola puede también materializarse en forma de bloque de posiciones reservadas. El bloque continuo tiene la ventaja de que se eliminan punteros y se acelera la recuperación y el inconveniente de que hay que reservar un bloque bastante grande para acomodar la estructura de tamaño más desfavorable que se prevea. También es difícil, o poco práctico, insertar y extraer elementos del interior del bloque. Como la memoria es casi siempre un recurso escaso, los bloques suelen reservarse para estructuras de tamaño fijo o que exijan la máxima velocidad de recuperación, como la pila.

LISTA CIRCULAR

Se llama lista circular a una lista encadenada en la que la última entrada señala hacia la primera (véase figura 9.6). Normalmente se lleva también un puntero del *bloque en curso*. Cuando se trata de sucesos o programas, el puntero de *suceso en curso* se mueve una posición a la derecha o a la izquierda en cada ocasión. La lista circular suele utilizarse cuando todos los bloques se supone que tienen idéntica prioridad, aunque también puede emplearse como caso particular de otras estructuras para facilitar la recuperación del primer bloque después del último cuando se lleva a cabo una búsqueda.



Figura 9.6
Lista circular.

Los programas de muestreo funcionan, por lo general, de forma parecida a una lista circular: interrogan a todos los periféricos, y cuando llegan al último vuelven a empezar por el primero.

ARBOLES

Siempre que hay una relación entre todos los elementos de una estructura (es lo que se llama sintaxis) puede utilizarse la estructura de árbol, que es de tipo descendente o genealógico

Figura 9.7
Arbol genealógico.

Figura 9.8
Lista doblemente encadenada

idad, como subrutinas
la misma no es una lista
.

. BLOQUE

arse en forma de bloque
ntinuo tiene la ventaja de
ra la recuperación y el
un bloque bastante gran-
amaño más desfavorable
poco práctico, insertar y
ue. Como la memoria es
loques suelen reservarse
exijan la máxima veloci-

encadenada en la que la
(véase figura 9.6). Nor-
ro del *bloque en curso*.
gramas, el puntero de
la derecha a la
lar suele utilizarse cuan-
enen idéntica prioridad,
caso particular de otras
del primer bloque des-
o una búsqueda.



nan, por lo general, de
interrogan a todos los
uelven a empezar por el

todos los elementos de
xis) puede utilizarse la
endente o genealógico.

La figura 9.7 ilustra una situación de este tipo: Jaime tiene un hijo llamado Roberto y una hija llamada Juana; ésta, a su vez, tiene tres niños: Elisa, Tomás y Felipe; Tomás, por su parte, tiene otros dos: Manuel y Cristina; Roberto (a la izquierda de la figura), por el contrario, no tiene descendencia.

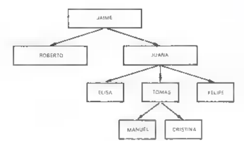


Figura 9.7
Árbol genealógico.

La estructura que une esas relaciones es un árbol. La figura 9.2, que ya hemos examinado al principio de este capítulo, es también un ejemplo de árbol sencillo. La estructura de directorio es un árbol de dos dimensiones. Los árboles se emplean siempre que los elementos pueden clasificarse de acuerdo con una estructura fija, que facilita la inserción y la recuperación de los mismos. Además, el árbol permite establecer grupos de información estructurados que pueden ser necesarios en ulteriores tratamientos (por ejemplo, en un compilador o en un intérprete).

LISTA DOBLEMENTE ENCADENADA

Entre los elementos de una lista pueden establecerse enlaces adicionales, de los que los más sencillos son los que aparecen en la llamada lista doblemente encadenada, que muestra la figura 9.8. Como se ve, contiene la estructura habitual de enlaces de izquierda a derecha más otra de derecha a izquierda. De lo que se trata es de facilitar la recuperación de los elementos situados justamente antes y después del que se está procesando, aunque para ello es necesario prever un puntero más por bloque.



Figura 9.8
Lista doblemente encadenada.

Búsqueda y ordenación

La búsqueda y la ordenación de los elementos de una lista es una operación que depende directamente de la estructura utilizada para dicha lista. Se han desarrollado numerosos algoritmos de búsqueda para las estructuras de datos habituales, de los que ya hemos empleado el direccionamiento indexado. Es un método de acceso posible cuando los elementos de la tabla están ordenados en función de un criterio conocido; tales elementos pueden recuperarse por sus números.

Se llama *búsqueda secuencial* a la exploración lineal de un bloque completo. Se trata de un método claramente ineficaz, pero que, en ocasiones, hay que emplearlo a falta de algo mejor si los elementos están totalmente desordenados.

La llamada *búsqueda binaria* o *logarítmica* busca un elemento en una lista ordenada dividiéndola en dos tras cada exploración. Si, por ejemplo, se trata de una lista alfabética, la búsqueda podría comenzar por determinar si el nombre buscado está antes o después de la mitad de la misma; si está después, se elimina la primera mitad de la lista y se vuelve a dividir en dos la segunda, para repetir en ella la misma operación, y así sucesivamente hasta llegar al elemento buscado; de esta forma, la longitud máxima que hay que explorar es $\log_2 n$, siendo n el número de elementos de la tabla. Aparte de ésta, hay muchas otras técnicas de búsqueda.

Resumen de la sección

En esta sección hemos tratado únicamente de hacer una breve presentación de las estructuras de datos que más frecuentemente utiliza el programador. Aunque tales estructuras están organizadas por tipos y tienen un nombre, la organización general de los datos en sistemas complejos recurre frecuentemente a combinaciones de varias de ellas u obliga al programador a inventar otras nuevas adecuadas al fin perseguido; así, las posibilidades sólo están limitadas por la imaginación del programador. De la misma manera, se han desarrollado una serie de técnicas de ordenación y búsqueda para las estructuras de datos más habituales, aunque su descripción está fuera del alcance de este libro. La finalidad de esta sección es subrayar la importancia que tiene el diseño de estructuras adecuadas para los datos que van a manipularse y proporcionar las herramientas necesarias para satisfacer ese objetivo. A continuación veremos ejemplos más detallados de su aplicación a programas reales.

PARTE II: EJ

Introducción

PARTE II: EJEMPLOS PRACTICOS

Introducción

Veremos aquí algunos ejemplos reales de diseño de estructuras de datos típicas: tabla, lista clasificada y lista encadenada, junto con los algoritmos de búsqueda, inserción y borrado correspondientes a las mismas.

Al lector interesado en técnicas avanzadas de programación le resultará de utilidad el estudio minucioso de los programas reunidos en esta sección.

Por el contrario, el principiante puede prescindir de su estudio en un principio y volver sobre ello cuando se considere más preparado.

Para seguir los ejemplos de esta parte es imprescindible entender perfectamente los conceptos presentados en la anterior. Además, los programas utilizarán todos los modos de direccionamiento del Z80 e integrarán muchas de las ideas y técnicas estudiadas en anteriores capítulos.

Estudiaremos aquí una lista sencilla, una alfabética y una lista encadenada con directorio. Para cada estructura vamos a desarrollar tres programas: buscar, introducir y eliminar.

Representación de datos en la lista

Tanto en la lista sencilla como en la alfabética representaremos los elementos de la misma forma:

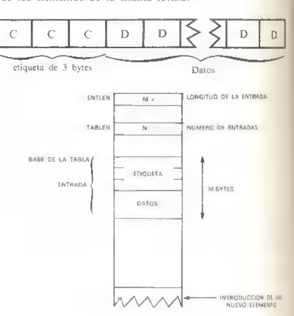


Figura 9.9
Estructura de la tabla.

Figura 9.10
Entradas típicas de la lista e
memoria.

Cada elemento o "entrada" consta de una etiqueta de 3 bytes y un bloque de datos de n bytes, estando n comprendido entre 1 y 253, de manera que cada uno utiliza, como máximo, una página (256 bytes). Todos los elementos de la lista tienen la misma longitud (véase figura 9.10). Los programas que manipulan estas dos sencillas listas comparten una serie de convenciones:

ENTLEN es la longitud de un elemento. Así, si cada uno tiene 10 bytes de datos, ENTLEN vale: 3 + 10 = 13.
TABASE es la base de la lista o tabla de memoria.
POINTR es el apuntador móvil del elemento en curso.
OBJETO es la entrada en curso de localización, inserción o eliminación.
TABLA es el número de entradas.

Lista sencilla

ELEMENTO 1
C
C
C
D
ETIQUETA
DATOS
D
ENTLEN
ELEMENTO 2
C
C
C
D
ETIQUETA
DATOS
D
ENTLEN

Figura 9.10
Entradas típicas de la lista en memoria.

Se supone que todas las etiquetas son diferentes. De todas formas, bastan modificaciones mínimas de los programas para cambiar estas convenciones.

Lista sencilla

La lista se organiza como tabla de n elementos no claisifcados (véase figura 9.11). Para buscar uno hay que recorrer la lista hasta dar con él o hasta llegar al final de aquélla. La inserción se realiza añadiendo nuevos elementos a los ya existentes. Cuando se elimina alguno, los situados en las posiciones superiores de la memoria —en caso de que haya alguno— se desplazan para cubrir los huecos y que la tabla sea continua.

BUSQUEDA

Se utiliza una técnica de búsqueda en serie que consiste en comparar las etiquetas una por una y letra por letra con la de OBJETO.

El puntero móvil POINTR se inicializa al valor de TABASE. El diagrama de flujo del algoritmo de búsqueda, que avanza de forma obvia, aparece en la figura 9.12. El programa se encuentra en la figura 9.16, situada al final de esta sección (programa "BUSCA"). La figura 9.17 contiene un pase de prueba del mismo.

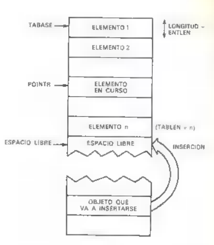


Figura 9.11
Lista sencilla.

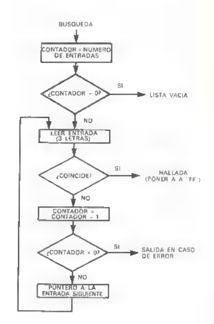


Figura 9.12
Diagrama de flujo de la búsqueda en la tabla.

Figura 9.13
Diagrama de flujo de inserci
en la tabla.



LISTA VACIA

HALLADA (PONER A A "FF")

SALIDA EN CASO DE ERROR

INSERCION

Para insertar un nuevo elemento se utiliza el primer bloque de bytes disponible en memoria (ENTLEN) al final de la lista (véase figura 9.11).

El programa empieza por comprobar [redacted] la nueva inserción no está ya en la lista (en este ejemplo se supone que todas las etiquetas son diferentes). Si no está, incrementa la longitud de la lista TABLEN y lleva OBJETO al final de la misma. El correspondiente diagrama de flujo aparece en la figura 9.13.

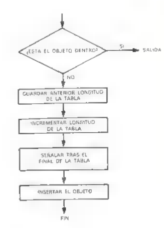


Figura 9.13
Diagrama de flujo de inserción en la tabla.

El programa completo aparece en la figura 9.16; [redacted] llama "NUEVO" y reside en las posiciones de memoria 013D a 0166.

El registro de índice IY señala la fuente. HL y DE son los apuntadores de destino.

ELIMINACION

Para eliminar un elemento de la lista no hay más que subir una posición los situados a continuación en direcciones superiores y decrementar la longitud de la lista. La operación se ilustra en la figura 9.14.

El programa es bastante sencillo, y aparece en la figura 9.16. Se llama "BORRAR" y reside en las direcciones de memoria 0167 [redacted] 018F; el diagrama de flujo está en la figura 9.15.

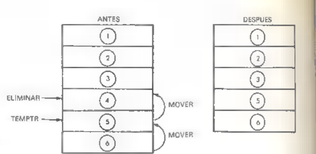


Figura 9.14
Eliminación de un elemento (tabla sencilla).

La posición de memoria TEMPTR es un puntero provisional que señala el elemento que ha de moverse hacia arriba.

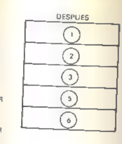
Durante la transferencia, POINTR señala siempre el hueco de la lista, es decir, el destino de la transferencia de bloque siguiente.

La bandera Z se usa a la salida para indicar que el resultado de la eliminación ha sido positivo.

Obsérvese que la instrucción LDIR ejecuta una transferencia de bloque automática eficaz (véase la dirección 0178 en la figura 9.16).

```
        LD    A, B           CONTADOR DE
                             BLOQUE
BLONUE  LD    BC, (ENTLEN)   LONGITUD DEL
                             BLOQUE
        LDIR
        DEC   A
        JP    NZ, BLONUE
```

Figura 9.15
Diagrama de flujo de la elimi-
nación de la tabla.



R es un puntero provi-
de moverse hacia arriba.
señala siempre el hueco
transferencia de bloque

a indicar que el resulta-

ejecuta una transferencia
rección 0178 en la figura

CONTADOR DE BLOQUE
LONGITUD DEL BLOQUE

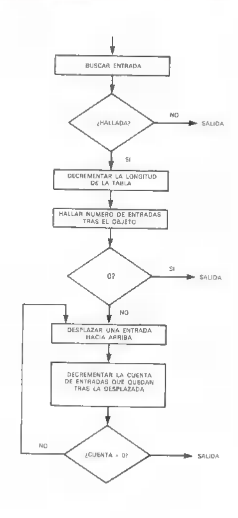


Figura 9.15
Diagrama de flujo de la eliminación de la tabla.

```
0100                10        ORG  #0100
0100 8F01           20 ENTLEN DEFW ENDER
0102 9101           30 TABLEN DEFW ENDER+2
0104 9201           40 TABASE DEFW ENDER+3
0106 9401           50 TEMP   DEFW ENDER+5
                    60 ;
0108 1600           70 BUSCA  LD   D,0          ;INICIALIZA [illegible]
010A 3A0201         80        LD   A,(TABLEN)   ;COMPRUEBA TABLA LONGITUD 0
010D A7             90        AND  A            ;DEFINE FLAGS
010E C8            100        RET  Z
010F 47            110        LD   B,A          ;ALMACENA LONGITUD TABLA
0110 DD2A0401      120        LD   IX,(TABASE)  ;PONE DIRECCION BASE EN IX
0114 DD7E00        130 BUCLE  LD   A,(IX+0)     ;COMPRUEBA PRIMERA LETRA
0117 FDBE00        140        CP   (IY+0)
011A C22F01        150        JP   NZ,SIGUIE
011D DD7E01        160        LD   A,(IX+1)     ;COMPRUEBA SEGUNDA LETRA
0120 FDBE01        170        CP   (IY+1)
0123 C22F01        180        JP   NZ,SIGUIE
0126 DD7E02        190        LD   A,(IX+2)     ;COMPRUEBA TERCERA LETRA
0129 FDBE02        200        CP   (IY+2)
012C CA3A01        210        JP   Z,ENCONT     ;SALIDA SI TODAS LETRAS
                   220 ;                        ;VALIDAS
012F 05            230 SIGUIE DEC  B            ;DECREMENTA CONTADOR
                   240 ;                        ;LONGITUD TABLA
0130 C8            250        RET  Z            ;SALIDA SI FINAL DE TABLA
0131 ED5B0001      260        LD   DE,(ENTLEN)  ;DEFINE EN IX LA SIGUIENTE
                   270 ;                        ;DIRECCION DE ENTRADA
0135 DD19          280        ADD  IX,DE
0137 C31401        290        JP   BUCLE        ;PRUEBA OTRA VEZ
013A 16FF          300 ENCONT LD   D,#0FF       ;FIJA EN [illegible] LA DIRECCION
                   310 ;                        ;DE ENTRADA EN LA TABLA
013C C9            320        RET  ;            ;CONTENIDA [illegible] IX
                   330 ;
                   340 ;
                   350 ;
013D CD0801        360 NUEVO  CALL BUSCA        ;VE SI EL OBJETO ESTA ALLI
0140 14            370        INC  D
0141 CA6601        380        JP   Z,FUERAE     ;SI D ERA [illegible] SALIR
0144 3A0201        390        LD   A,(TABLEN)
0147 5F            400        LD   E,A          ;CARGA [illegible] CON LONGITUD TABLA
0148 3C            410        INC  A
0149 320201        420        LD   (TABLEN),A   ;INCREMENTA LONGITUD TABLA
014C 1600          430        LD   D,0
014E 2A0401        440        LD   HL,(TABASE)
0151 ED4B0001      450        LD   BC,(ENTLEN)  ;FIJA B A LA LONGITUD DE
                   460 ;                        ;UNA ENTRADA
0155 41            470        LD   B,C
0156 19            480 BUCLEE ADD  HL,DE
0157 10FD          490        DJNZ BUCLEE       ;SUMA HL A (ENTLEN*TABLE)
0159 ED4B0001      500        LD   BC,(ENTLEN)
015D FDE5          510        PUSH IY           ;MUEVE IY [illegible] DE
015F D1            520        POP  DE
0160 EB            530        EX   DE,HL
0161 EDB0          540        LDIR ;            ;MUEVE LA MEMORIA DEL
                   550 ;                        ;OBJETO        AL FINAL
0163 01FFFF        560        LD   BC,#0FFFF    ;DE LA TABLA
0166 C9            570 FUERAE RET
                   580 ;
                   590 ;
                   600 ;
0167 CD0801        610 BORRAR CALL BUSCA        ;LOCALIZA ENTRADA A BORRAR
016A 14            620        INC  D            ;VE SI SE HA ENCONTRADO
016B C28B01        630        JP   NZ,SALIDA
016E 3A0201        650        LD   A,(TABLEN)   ;DECREMENTA LONGITUD TABLA
0171 3D            660        DEC  A
0172 320201        670        LD   (TABLEN),A
0175 05            680        DEC  B            ;AHORA B=NUMERO DE ENTRADAS
                   690 ;                        ;DEJADAS EN LA TABLA
0176 CA8B01        700        JP   Z,SALIDA     ;...DESPUES DE BORRAR UNA
0179 DDE5          710        PUSH IX           ;MUEVE IX [illegible] DE
017B [illegible]   720        POP  DE
017C 2A0001        730        LD   HL,(ENTLEN)  ;COLOCA HL UNA ENTRADA
                   740 ;                        ;DELANTE DE DE
017F 19            750        ADD  HL,DE
0180 78            760        LD   A,B          ;FIJA EL CONTADOR DE BLOQUE
0181 ED4B0001      770 BLONUE LD   BC,(ENTLEN)  ;FIJA EL CONTADOR DE
```

Figura 9.16
Programas de la lista sencilla.

Figura 9.16
Programas de la lista sencilla (continuación).

Figura 9.17
Pase de prueba de los prog mas de la lista sencilla.

```
;INICIALIZA D
;COMPRUEBA TABLA LONGITUD 0
;DEFINE FLAGS

;ALMACENA LONGITUD TABLA
;PONE DIRECCION BASE EN IX
;COMPRUEBA PRIMERA LETRA

;COMPRUEBA SEGUNDA LETRA

;COMPRUEBA TERCERA LETRA

;SALIDA ■ TODAS LETRAS
;VALIDAS
;DECREMENTA CONTADOR
;LONGITUD TABLA
;SALIDA SI FINAL DE TABLA
;DEFINE EN IX LA SIGUIENTE
;DIRECCION DE ENTRADA

;PRUEBA OTRA VEZ
;FIJA EN ■ LA DIRECCION
;DE ENTRADA EN LA TABLA
;CONTENIDA EN IX

;VE SI EL OBJETO ESTA ALLI

;SI ■ ERA FF SALIR

;CARGA E CON LONGITUD TABLA

;INCREMENTA LONGITUD TABLA

;FIJA B A LA LONGITUD DE
;UNA ENTRADA

;SUMA HL ■ (ENTLEN*TABLE)

;MUEVE IY A DE

;MUEVE LA MEMORIA DEL
;OBJETO        AL FINAL
;DE LA TABLA

;LOCALIZA ENTRADA A BORRAR
;VE ■ SE HA ENCONTRADO

;DECREMENTA LONGITUD TABLA

;AHORA B=NUMERO DE ENTRADAS
;DEJADAS EN LA TABLA
;...DESPUES DE BORRAR UNA
;MUEVE IX A DE

 ;COLOCA HL UNA ENTRADA
;DELANTE DE DE

;FIJA EL CONTADOR DE BLOQUE
;FIJA EL CONTADOR DE
```

```
                  780 ;                          ;LONGITUD DE BLOQUE
0185 ED80         790          LDIR ;            ;SHIFT DE UNA ENTRADA DE
                  800 ;                          ;LA TABLA
0187 3D           810          DEC  A
0188 C28101       820          JP   NZ,BLONUE    ;SHIFT DE OTRO BLOQUE
018B 01FFFF       830 SALIDA   LD   BC,#OFFFF    ;DEMUESTRA QUE ESTA HECHO
018E C9           840 FUERA    RET
                  850 ;
018F              860 ENDER    END

BLONUE 0181     BORRAR 0167          NUEVO  0130     OFFFF  018B
BUCLE  0114     BUCLEE 0156          SALIDA 018B     SIGUIE 012F
BUSCA  0108     ENCONT 013A          TABASE 0104     TABLEN 0102
ENDER  018F     ENTLEN 0100          TEMP   0106
FUERA  018E     FUERAE 0166
```

Figura 9.16
Programas de la lista sencilla (continuación).

Distribución de ■ memoria

Lista de objetos con sus posiciones en memoria

```
 DM300
0300  53 4F 4E 31 31 31 31 31-31 31 31 31 31 00 00 00   SON1111111111...
0310  44 41 44 32 32 32 32 32-32 32 32 32 32 00 00 00   DAD2222222222...
0320  4D 4F 4D 33 33 33 33 33-33 33 33 33 33 00 00 00   MOM3333333333...
0330  55 4E 43 34 34 34 34 34-34 34 34 34 34 00 00 00   UNC4444444444...
0340  41 4E 54 35 35 35 35 35-35 35 35 35 35 00 00 00   ANT5555555555...
0350  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0360  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0370  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
```

```
-SY
Y=0000 300
```

Llevar IY a 0300H (puntero a OBJETO)

```
-G193/196
P=0196 0196
```

Prueba de (inserción)

Configuración de la tabla tras el pase del programa

```
-DM400
0400  53 4F 4E 31 31 31 31 31-31 31 31 31 31 00 00 00   SON1111111111...
0410  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0420  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0430  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0440  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0450  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0460  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0470  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
```

```
-SY
Y=0300 310
```

Llevar IY a 0310H (siguiente OBJETO)

```
-G193/196
P=0196 0196
```

Pase de "NUEVO" (inserción)

Configuración de la tabla tras la segunda inserción

```
-DM400
0400  53 4F 4E 31 31 31 31 31-31 31 31 31 31 44 41 44   SON1111111111DAD
0410  32 32 32 32 32 32 32 32-32 32 00 00 00 00 00 00   2222222222......
0420  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0430  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0440  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0450  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0460  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0470  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
```

. . . (Más inserciones) . . .

Configuración de la tabla tras varias inserciones

```
-DM400
0400  53 4F 4E 31 31 31 31 31-31 31 31 31 31 44 41 44   SON1111111111DAD
0410  32 32 32 32 32 32 32 32-32 32 55 4E 43 34 34 34   2222222222UNC444
0420  34 34 34 34 34 34 34 4D-4F 4D 33 33 33 33 33 33   4444444MOM333333
0430  33 33 33 33 41 4E 54 35-35 35 35 35 35 35 35 35   3333ANT555555555
0440  35 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   5...............
0450  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0460  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0470  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
```

Figura 9.17
Pase de prueba de los programas de la lista sencilla.

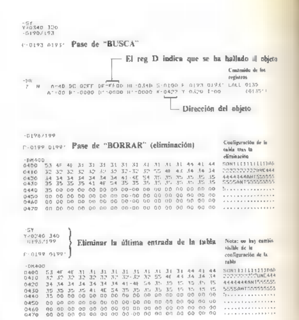


Figura 9.17
Pase de prueba de los programas de la lista sencilla (continuación).

Lista alfabética

Esta lista o "tabla", a diferencia de la anterior, mantiene todos sus elementos ordenados alfabéticamente, lo que permite utilizar técnicas de búsqueda más rápidas que la lineal; en este caso utilizaremos la búsqueda binaria.

BUSQUEDA

El algoritmo es el clásico de búsqueda binaria. La técnica es básicamente similar a la que se usa para buscar un nombre en una guía telefónica: se abre hacia la mitad y, según lo que ponga allí, se avanza o se retrocede para acercarse al nombre buscado; es un método rápido y bastante fácil de realizar.

El diagrama de flujo aparece en la figura 9.18, y el programa en la 9.23.

Los elementos están ordenados en la lista alfabéticamente y se recuperan mediante la técnica binaria o logarítmica, como indica el ejemplo de la figura 9.19. El procedimiento es un tanto complicado, porque es preciso llevar la cuenta de varias condiciones. El problema más importante es evitar la búsqueda de algo que no está en la lista, ya que en tal caso el programa escrutaría incesantemente los elementos situados justo antes y después del buscado; para evitarlo, se mantiene en el programa una bandera que conserva el valor de la de acarreo tras una búsqueda infructuosa. Cuando el valor INCMNT, que indica la magnitud en que debe incrementarse a continuación el puntero, alcanza el valor "1", se activa otra bandera llamada "CERRAR" al valor de la bandera COMPR; de esta forma, como todos los incrementos ulteriores serán de "1", si el puntero sobrepasa el punto en que debería estar el objeto, COMPR deja de ser igual a CERRAR, y la búsqueda termina. Este recurso permite, además, a la rutina NUEVO determinar la situación de los punteros lógicos y físicos en relación con el lugar al que irá el objeto.

Por tanto, si el OBJETO buscado no está en la tabla y el puntero en curso se incrementa en uno, queda activada la bandera CERRAR. Cuando la rutina avance el paso siguiente, el resultado de la comparación será el contrario del anterior, las dos banderas dejarán de coincidir y el programa se dirigirá a la salida con la indicación "no hallado".

El otro problema importante que debe resolverse es la posibilidad de salirse de la tabla al sumar o restar el valor de incremento; para evitarlo se lleva a cabo una "suma" o una "resta" de prueba con el puntero lógico y el valor de longitud que registra el número real de elementos, no las posiciones físicas de memoria utilizadas por los punteros físicos.

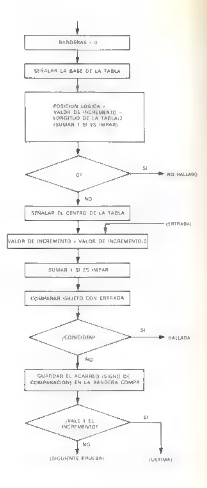
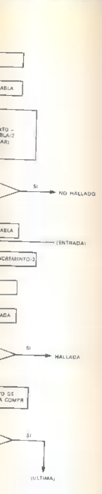


Figura 9.18
Diagrama de flujo de la búsqueda binaria.

Figura 9.18
Diagrama de flujo de la b
queda binaria (continuación



(SIGUIENTE PRUEBA)
(ULTIMA)
¿CERRAR = 0?
SI
CERRAR = COMPR
NO
NO HALLADA
NO
¿CERRAR = COMPR?
SI
RESTAR FF
SALTO A COMPR
SUMAR 1
¿SE SALDRÁ EL INCREMENTO DE LA TABLA?
SI
¿FINAL DE LA TABLA?
SI
NO HALLADA
NO
(DEALTO)
NO
ACTUALIZAR PUNTEROS
(ENTRADA)
SUBIR 1 LOS PUNTEROS
SI
¿SE SALDRÁ EL INCREMENTO DE LA TABLA?
NO
NO HALLADA
SI
¿FINAL DE LA TABLA?
ACTUALIZAR PUNTEROS
(ENTRADA)
NO
(DEBAJO)
BAJAR 1 LOS PUNTEROS
INCREMENTO = 1
CERRAR = COMPR
(ENTRADA)

Figura 9.18
Diagrama de flujo de la búsqueda binaria (continuación).

```
(0121)    LD    A,  C
          SRL   A
          ADC   0
          LD    C,  A
```

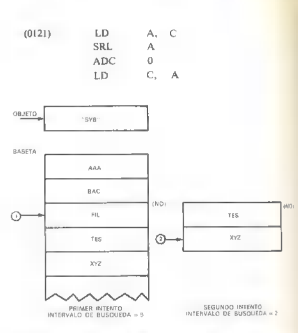


Figura 9.19
Búsqueda binaria.

En resumidas cuentas, el programa emplea dos banderas para memorizar la información: COMPR y CERRAR. La primera mantiene el valor "0" ó "1" del acarreo tras la última comparación, lo que determina si el elemento en prueba [illegible] mayor o menor que el verificado inmediatamente antes; C señala la relación: si es "1", quiere decir que el elemento es menor que el objeto, y COMPR se pone a "1"; [illegible] es "0", significa que el elemento es mayor que el objeto, y COMPR se pone a "FF".

La bandera CERRAR se hace igual a COMPR cuando el incremento de búsqueda INCMNT alcanza el valor "1"; si al paso siguiente COMPR y CERRAR dejan de ser iguales, [illegible] que el elemento buscado no se encuentra.

El programa utiliza también las variables siguientes:

- LOGPOS indica la posición lógica en la tabla (número de elemento).
- INCMNT representa el valor en que hay que incrementar o decrementar el puntero en curso si falla la comparación siguiente.

LONTAB, como es habitual, representa la longitud total de la lista.

LOGPOS e INCMNT se comparan con TABLA, para garantizar que no se sobrepasan los límites de la lista.

El programa, llamado "BUSQ", aparece en la figura 9.23, y reside en las posiciones de memoria 010A a 01D9. Debe estudiarse con atención, porque ■ mucho más complicado que el de búsqueda lineal.

Como el intervalo de búsqueda puede ser par o impar, es preciso introducir una corrección (en efecto, el programa no puede señalar el elemento central de una lista de cuatro); si es impar, se emplea un "truco" muy sencillo para dirigir el puntero hacia el elemento central: la división por 2 va acompañada de un desplazamiento a la derecha, que se hace sumando a dicho puntero el "1", el cual pasa al acarreo tras aplicar la instrucción SRL a un intervalo impar.

A continuación se compara OBJETO con el elemento central del nuevo intervalo de búsqueda; si la comparación es positiva, el programa termina. En caso contrario ("NOBUEN") se reinicia ■ "0" el acarreo para indicar que OBJETO es menor que el elemento. Cuando INCMNT alcanza el valor "1", se verifica la bandera CERRAR, que se había inicializado a "0", para ver ■ está activada, y se activa en caso negativo; si ya está activada, se procede ■ una verificación para determinar si se ha pasado la posición en que debería estar OBJETO sin encontrarlo.

Cuando el acarreo vale "1", el puntero señala el elemento situado por debajo de OBJETO.

INSERCION DE UN ELEMENTO

Para insertar un nuevo elemento se lleva a cabo una búsqueda binaria. Si ya se encuentra en la tabla, es que no hay que introducirlo (supondremos que todos los elementos son distintos). Si no aparece, se inserta inmediatamente antes o después del último elemento comparado, según lo indicado por la bandera COMPR. Todos los elementos que siguen al nuevo deben descender la posición de un bloque para hacerle sitio.

El método de inserción se ilustra en la figura 9.20 y el correspondiente programa aparece en la 9.23. Se llama NUEVO, y empieza en la posición de memoria 01DA. Obsérvese que se han empleado las instrucciones automáticas del Z80 LDDR y LDIR para hacer transferencias de bloques eficaces.

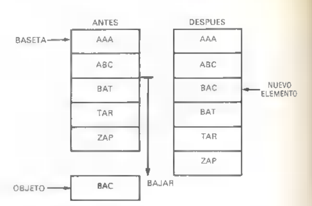


Figura 9.20
Inserción de "BAC"

ELIMINACION DE UN ELEMENTO

Como antes, se empieza por una búsqueda binaria para encontrar el objeto. Si falla, es que no se encuentra en la lista y no puede eliminarse. Si aparece, se elimina, y todos los que le siguen ascienden una posición, como se ve en el ejemplo de la figura 9.21. El programa aparece en la 9.23, y el diagrama de flujo, en la 9.22. Se llama "BORRAR" y reside en la dirección 0221.

La figura 9.24 recoge un pase de prueba de todos los programas propuestos.

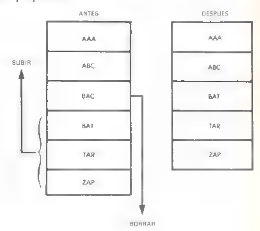
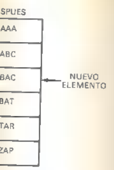


Figura 9.21
Eliminación de "BAC".

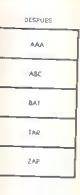
Figura 9.22
Diagrama de flujo de elimi-
ción (lista alfabética)



O

búsqueda binaria para
encuentra en la lista y
na. y todos los que le
ve en el ejemplo de la
9.23. y el diagrama de
reside en la dirección

eba de todos los pro-



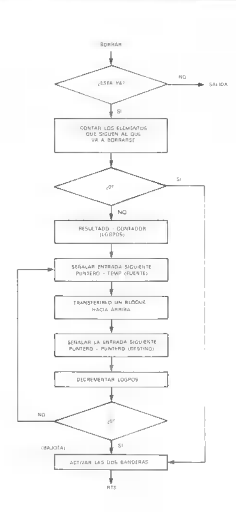


Figura 9.22
Diagrama de flujo de eliminación (lista alfabética).

```
0100                   ORG   #0100
0100 5402       CERRAR DEFW  FIN
0102 5502       COMPR  DEFW  FIN+1
0104 5602       LONTAB DEFW  FIN+2
0106 5702       BASETA DEFW  FIN+3
0108 5902       LONENT DEFW  FIN+5
                ;
010A 3E00       BUSO   LD    A,0
010C 320001            LD    (CERRAR),A    ;POSICIONES DE FLAG CERO
010F 320201            LD    (COMPR),A
0112 57                LD    D,A
0113 2A0601            LD    HL,(BASETA)   ;INICIALIZA HL
0116 3A0401            LD    A,(LONTAB)
0119 CB3F              SRL   A             ;DIVIDE POR 2
011B CE00              ADC   A,0
011D 4F                LD    C,A           ;ALMACENA VALOR INCREMENTO
011E 47                LD    B,A           ;ALMACENA VALOR DE
                ;                          ;POSICION LOGICA
011F CAC401            JP    Z,NOVALE      ;COMPRUEBA SI LOGITUD ES 0
0122 5F                LD    E,A           ;MULTIPLICA (E-1)*LONENT
0123 1D                DEC   E
0124 CDC701            CALL  MULT
0127 19                ADD   HL,DE         ;COLOCA HL EN MITAD TABLA
0128 E5         ENTRAD PUSH  HL            ;CARGA HL EN IX
0129 DDE1              POP   IX
012B 79                LD    A,C           ;DIVIDE EL VALOR DE
                ;                          ;INCREMENTO POR 2
012C CB3F              SRL   A
012E CE00              ADC   A,0
0130 4F                LD    C,A
0131 DD7E00            LD    A,(IX+0)      ;COMPARA LA PRIMERA LETRA
0134 FDBE00            CP    (IY+0)
0137 C24C01            JP    NZ,NOBUEN
013A DD7E01            LD    A,(IX+1)      ;COMPARA LA SEGUNDA LETRA
013D FDBE01            CP    (IY+1)
0140 C24C01            JP    NZ,NOBUEN
0143 DD7E02            LD    A,(IX+2)      ;COMPARA LA TERCERA LETRA
0146 FDBE02            CP    (IY+2)
0149 CAC601            JP    Z,VALE
014C 3E01       NOBUEN LD    A,1           ;FIJA FLAG DE RESULTADO DE
                ;                          ;LA COMPARACION AL
014E DA5301            JP    C,TESTS       ;..RESULTADO DE ELLA (1,FF)
0151 3EFF              LD    A,#FF
0153 320201     TESTS  LD    (COMPR),A
0156 79                LD    A,C           ;ES 1 VALOR DE INCREMENTO ?
0157 3D                DEC   A
0158 C27301            JP    NZ,SIGTES
015B 3A0001            LD    A,(CERRAR)
015E A7                AND   A
015F CA6D01            JP    Z,NOCERR
0162 57                LD    D,A
0163 3A0201            LD    A,(COMPR)
0166 92                SUB   D
0167 C27301            JP    NZ,SIGTES
016A C3C401            JP    NOVALE
016D 3A0201     NOCERR LD    A,(COMPR)     ;FIJA FLAG DE CIERRE A LA
                ;                          ;DIRECCION DE
0170 320001            LD    (CERRAR),A    ;BUSQUEDA PARA PREVENIR UNA
                ;                          ;REPETICION
0173 DDE5       SIGTES PUSH  IX            ;PREPARA HL Y DE PARA SUMAR
0175 E1                POP   HL            ;O RESTAR EL VALOR DE INCR.
0176 59                LD    E,C
0177 CDC701            CALL  MULT
017A 3A0201            LD    A,(COMPR)     ;COMPRUEBA SI QUIERE SUMAR
                ;                          ;O RESTAR
017D 3C                INC   A
017E C2A001            JP    NZ,SUMALO
0181 78                LD    A,B           ;COMPRUEBA SI AL RESTAR
0182 91                SUB   C             ;TENDRA UN VALOR DEMASIADO
                ;                          ;BAJO
0183 CABF01            JP    Z,DEBAJO
0186 DABF01            JP    C,DEBAJO
0189 47                LD    B,A           ;FIJA EL VALOR DE LA NUEVA
                ;                          ;POSICION LOGICA
018A ED52              SBC   HL,DE         ;CAMBIA LA MISMA DIRECCION
018C C32801            JP    ENTRAD
```

Figura 9.23
Programa de búsqueda binaria.

Figura 9.23
Programa de búsqueda bin
(continuación).

```
;POSICIONES DE FLAG CERO

;INICIALIZA HL

;DIVIDE POR 2

;ALMACENA VALOR INCREMENTO
;ALMACENA VALOR DE
;POSICION LOGICA
;COMPRUEBA SI LOGITUD ES ■
;MULTIPLICA (E-1)*LONENT

;COLOCA HL EN MITAD TABLA
;CARGA HL EN IX

;DIVIDE EL VALOR DE
;INCREMENTO POR 2

COMPARA LA PRIMERA LETRA

COMPARA LA SEGUNDA LETRA

COMPARA LA TERCERA LETRA

FIJA FLAG DE RESULTADO DE
LA COMPARACION AL
..RESULTADO DE ELLA (1,FF)

ES 1 VALOR DE INCREMENTO ?

FIJA FLAG DE CIERRE A LA
DIRECCION DE
BUSQUEDA PARA PREVENIR UNA
REPETICION
PREPARA HL Y DE PARA SUMAR
O RESTAR EL VALOR DE INCR.

OMPRUEBA SI QUIERE SUMAR
 RESTAR

OMPRUEBA SI AL RESTAR
ENDRA UN VALOR DEMASIADO
AJO

IJA EL VALOR ■ LA NUEVA
OSICION LOGICA
AMBIA LA MISMA DIRECCION
```

Figura 9.23
Programa de búsqueda binaria (continuación).

```
01BF 78        DEBAJO LD   A,B          ;VE SI LA POSICION ES UNO
0190 3D               DEC  A
0191 CAC401           JP   Z,NOVALE     ;SI ES ASI, SE VA
0194 ED5B0801         LD   DE,(LONENT)  ;SOLO RESTA UNA POSICION
0198 37               SCF
0199 3F               CCF
019A ED52             SBC  HL,DE
019C 05               DEC  ■            ;CAMBIA LA POSICION LOGICA
019D C3B901           JP   CREAL
01A0 3A0401    SUMALO LD   A,(LONTAB)   ;COMPRUEBA SI LA POSICION
01A3 90               SUB  B            ;ACTUAL MAS EL INCREMENTO
01A4 91               SUB  C            ;SON MAYORES QUE LONTAB
01A5 CAAF01           JP   C,DEALTO
01A8 19               ADD  HL,DE        ;SI NO ■ ASI CAMBIA LA
               ;                        ;DIRECCION ACTUAL
01A9 78               LD   A,B          ;CAMBIA EL VALOR ■ LA
               ;                        ;POSICION LOGICA
01AA 81               ADD  A,C
01AB 47               LD   B,A
01AC C32B01           JP   ENTRAD
01AF 81        DEALTO ADD  A,C          ;VE SI LA POSICION ■ AL
               ;                        ;PRINCIPIO DE LA
01B0 CAC401           JP   Z,NOVALE     ;TABLA (IGUAL A LONTAB-B)
01B3 ED5B0801         LD   DE,(LONENT)  ;SUMA UNA POSICION ENTRADA
01B7 19               ADD  HL,DE
01B8 04               INC  B            ;INCREMENTA POSICION LOGICA
01B9 0E01      CREAL  LD   C,1          ;FIJA A UNO EL INCREMENTO
01BB 3A0201           LD   A,(COMPR)    ;DEFINE FLAG DE CIERRE
01BE 320001           LD   (CERRAR),A   ;PARA COMPARAR RESULTADO
01C1 C32B01           JP   ENTRAD
01C4 16FF      NOVALE LD   D,#0FF
01C6 C9        VALE   RET
               ;
01C7 ■         MULT   PUSH HL           ;MULTIPLICA ■ POR EL VALOR
01C8 C5               PUSH BC           ;DE (LONENT) COLOCANDOLO
               ;                        ;EN DE
01C9 1600             LD   D,0
01CB 210000           LD   HL,0000
01CE ED4B0801         LD   BC,(LONENT)
01D2 41               LD   B,C
01D3 19        SUMA■  ADD  HL,DE
01D4 10FD             DJNZ SUMAS
01D6 C1               POP  BC
01D7 EB               EX   DE,HL
01D8 E1               POP  HL
01D9 C9               RET
               ;
               ;
               ;
01DA CD0A01    NUEVO  CALL BUSQ         ;VE SI OBJETO YA ESTA ALLI
01DD 14               INC  ■
01DE C22A02           JP   NZ,SALIR
01E1 3A0401           LD   A,(LONTAB)
01E4 A7               AND  A
01E5 CA1602           JP   Z,INSERT
01E8 3A0201           LD   A,(COMPR)
01EB 3C               INC  A
01EC CAF701           JP   Z,LADOAL
01EF ED5B0801         LD   DE,(LONENT)  ;COMPR=1, FIJA HL A DONDE
01F3 19               ADD  HL,DE        ;DEBE IR EL OBJETO
01F4 C3FB01           JP   SISTEM
01F7 05        LADOAL DEC  B            ;COMPR=0, ■ PARA RESTAR
01F8 3A0401    SISTEM LD   A,(LONTAB)   ;VE CUANTAS ENTRADAS SE HAN
               ;                        ;HECHO
01FB 90               SUB  B
01FC CA1602           JP   Z,INSERT
01FF 5F               LD   E,A          ;FIJA HL A LA ULTIMA
               ;                        ;POSICION DE LA ULTIMA
0200 CDC701           CALL MULT         ;ENTRADA
0203 19               ADD  HL,DE
0204 2B               DEC  HL
0205 EB               EX   DE,HL        ;DE ES UNA ENTRADA ANTES HL
0206 2A0801           LD   HL,(LONENT)
0209 19               ADD  HL,DE
020A ■                EX   DE,HL
020B ED4B0801  MOVIM  LD   BC,(LONENT)  ;SHIFT UNA ENTRADA DE MEM.
```

```
020F EDB8                    LDDR
0211 3D                      DEC  A
0212 C20B02                  JP   NZ,MOVIM      ;REPITE [redacted] NECESARIO
0215 23                      INC  HL
0216 FDE5            INSERT  PUSH IY            ;CARGA EN ESPACIO VACIO
0218 D1                      POP  DE
0219 EB                      EX   DE,HL
021A ED4B0801                LD   BC,(LONENT)
021E EDB0                    LDIR
0220 3A0401                  LD   A,(LONTAB)    ;INCREMENTA LONGITUD TABLA
0223 3C                      INC  A
0224 320401                  [redacted]   (LONTAB),A
0227 01FFFF                  LD   BC,#0FFFF     ;MUESTRA LO QUE HA HECHO
022A C9              SALIR   RET
                     ;
                     ;
                     ;
022B CD0A01          BORRAR  CALL BUSQ          ;COGE LA DIRECCION OBJETO
022E 14                      INC  D             ;VE SI OBJETO ESTA ALLI
022F CA5302                  JP   Z,SALIRE
0232 ED5B0801                LD   DE,(LONENT)
0236 EB                      EX   DE,HL
0237 19                      ADD  HL,DE         ;DE [redacted] POS. DE OBJETO, HL
0238 3A0401                  [redacted]   A,(LONTAB)    ;ESTA UNA ENTRADA ANTES
023B 90                      SUB  [redacted]             ;VE CUANTAS ENTRADAS SE HAN
023C CA4902                  JP   Z,BAJOTA      ;HECHO
023F ED4B0801        SHIFT   LD   BC,(LONENT)
0243 EDB0                    LDIR ;SHIFT ABAJO UNA LONENT
0245 3D                      DEC  A
0246 C23F02                  JP   NZ,SHIFT
0249 3A0401          BAJOTA  LD   A,(LONTAD)    ;DECREMENTA LONGITUD TABLA
024C 3D                      DEC  A
024D 320401                  LD   (LONTAB),A
0250 01FFFF                  LD   BC,#0FFFF     ;MUESTRA LO QUE HA REALIZ.
0253 C9              SALIRE  RET
                     ;
0254                 FIN     END


BAJOT[redacted] 0249    BASETA 0106       MULT   01C7    NOBUEN 014C
BORRAR 022B    BUSQ   010A       NOCERR 016D    NOVALE 01C4
CERRAR 0100    COMPR  0102       NUEVO  01DA    SALIR  022A
CREAL  0189    DEALTO 01AF       SALIRE 0253    SHIFT  023F
DEBAJO 01BF    ENTRAD 0128       SIGTES 0173    SISTEM 01F8
FIN    0254    INSERT 0216       SUMALO 01A0    SUMAS  01D3
LADOAL 01F7    LONENT 0108       TESTS  0153    VALE   01C6
LONTAB 0104    MOVIM  020B
```

Figura 9.23
Programa de búsqueda binaria (continuación).

Lista encadenada

Los elementos de esta lista contienen una etiqueta de tres caracteres alfanuméricos, 250 bytes de datos, un apuntador de dos bytes con la dirección de partida del elemento siguiente y un marcador de un byte; cuando éste vale "1", impide [redacted] la rutina de inserción colocar un elemento nuevo en el lugar de otro preexistente.

Además, para facilitar la recuperación, hay un directorio que contiene un apuntador orientado hacia la primera entrada de cada letra del alfabeto. Las etiquetas son caracteres alfabéticos ASCII. Todos los apuntadores del final de la lista tienen un valor NIL, que en este caso se ha elegido igual a la base de la tabla, ya que dicho valor nunca debe aparecer en el interior de la lista encadenada.

Figura 9.24
Pasa de prueba de la lista a
bética

```
                  ;REPITE SI ES NECESARIO
                  ;CARGA EN ESPACIO VACIO

                  ;INCREMENTA LONGITUD TABLA

                  ;MUESTRA LO QUE HA HECHO

                  ;COGE LA DIRECCION OBJETO
                  ;VE SI OBJETO ESTA ALLI

                  ;DE ■ POS. DE OBJETO, HL
                  ;ESTA UNA ENTRADA ANTES
                  ;VE CUANTAS ENTRADAS SE HAN
                   HECHO
                  NA LONENT

                  ;DECREMENTA LONGITUD TABLA

                  ;MUESTRA LO QUE HA REALIZ.

7     NOBUEN 014C
D     NOVALE 01C4
A     SALIR  022A
5     SHIFT  023F
3     SISTEM 01FB
D     SUMAS  01D3
5     VALE   01C6
```

una etiqueta de tres
tos. un apuntador de
elemento siguiente y
vale "1", impide a la
nuevo en el lugar de

hay un directorio que
a primera entrada de
caracteres alfabéticos
de la lista tienen un
igual ■ la base de la
recer en el interior de

Tabla inicial

```
-DM400
0400  [illegible]
0410  [illegible]
0420  [illegible]
0430  [illegible]
0440  [illegible]
0450  [illegible]
0460  [illegible]
0470  [illegible]
```

Relación de objetos y sus posiciones en memoria

```
-DM300
[illegible]
```

Pase de "NUEVO"

Tabla tras la inserción

```
-DM400
[illegible]
```

Pase de "NUEVO" con otro bojeto

Tabla tras la inserción. Nota: la tabla sigue siendo alfabética

```
-DM400
[illegible]
```

· · · (inserciones adicionales) · · ·

Configuración de la tabla tras insertar todos los objetos

```
-DM400
0400  41 4E 54 35 35 35 35 35-35 35 35 35 35 44 41 44   ANT5555555555DAD
0410  32 37 32 32 32 32 32 32-32 32 4D 4F 4D 33 33 33   2222222222MOM333
0420  33 33 33 33 33 33 33 53-4F 4E 31 31 31 31 31 31   3333333SON111111
0430  31 31 31 31 55 4E 43 34-34 34 34 34 34 34 34 34   1111UNC444444444
0440  34 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   4...............
0450  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0460  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0470  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................

-SY
Y=0340 300
-0260/263

P=0263 0263'
```

Pase de "BUSQ" de 'SON' (en la dirección 0300)

Hallado

```
-DR
 Z  N   A=4E BC=0401 DE=0000 HL=0427 S=0100 P=0263 0263' CALL 0100
        A'=00 B'=0000 D'=0000 H'=0000 X=0427 Y=0300 I=00       '0100')
```

Dirección del objeto en la tabla

(comprobar en la tabla de arriba que está 'SON')

Figura 9.24
Pase de prueba de la lista alfabética.

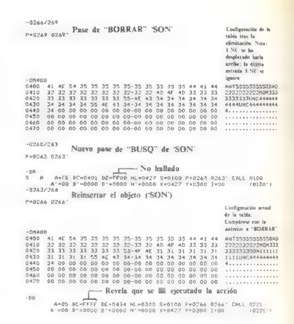


Figura 9.24
Pase de prueba de la lista alfabética (continuación).

Los programas de inserción y eliminación ejecutan las manipulaciones obvias de los apuntadores. Utilizan la bandera INDEXA para indicar [redacted] el que señala un objeto procede de un elemento anterior de la lista o del directorio. Los programas correspondientes se encuentran en la figura 9.29. Las estructuras de datos aparecen en la 9.25.

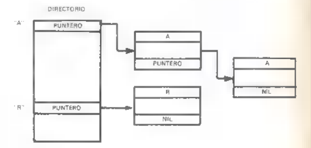
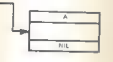


Figura 9.25
Estructura de la lista encadenada.



Una aplicación de esta estructura de datos sería un libro de direcciones informatizado, en el que cada persona estuviese representada por un código único de tres letras (sus iniciales, por ejemplo) y tuviese en el campo de datos una dirección simplificada y el número de teléfono (hasta 250 caracteres). Examinemos dicha estructura más de cerca. El formato de cada entrada es:



Las convenciones son las habituales:

LONENT: longitud total en bytes.
BASETA: dirección de la base de la lista.

Se supone que la dirección de OBJETO reside siempre en el registro IY antes de entrar al programa. REFBAS señala la dirección de la base del directorio o "tabla de referencia".

Cada una de las direcciones de dos bytes de éste apunta a la primera aparición de la letra a que corresponde en la lista, de manera que cada uno de los grupos de elementos que comparten la misma inicial en sus etiquetas forman, en realidad, una lista independiente dentro de la estructura general. Esta característica facilita la búsqueda, y es análoga a una agenda de direcciones. Obsérvese que durante la inserción y la eliminación no se mueve ningún dato, sino que únicamente se modifican los punteros, como en cualquier estructura de lista encadenada.

Si no aparece ninguna entrada con una inicial específica o si no hay ninguna que siga alfabéticamente a otra preexistente, sus punteros señalan al comienzo de la tabla (= "NIL"). Al fondo de ésta se conviene en almacenar un valor tal que el valor absoluto de la diferencia entre él y "Z" sea mayor que la diferencia entre "A" y "Z"; esto constituye el indicador de fin de tabla (EOT). En este caso, el EOT ocupa la misma cantidad de memoria que un elemento normal, pero si se desea puede tener un solo byte; las letras son caracteres alfabéticos en código ASCII. Si se modifica este extremo, habría que cambiar también la constante de la rutina PRETAB.

El marcador de fin de tabla se lleva al valor del principio de la misma ("NIL").

Se conviene en llevar los "punteros NIL" del final de una serie, o de una posición del directorio que no señale una serie, al valor de la base de la tabla, para disponer de una identificación única, aunque podría emplearse otra convención. En concreto, si se emplea un marcador diferente como EOT ■ ahorra algo de espacio, porque no es preciso mantener entradas NIL para los artículos inexistentes.

La inserción y la eliminación se llevan a cabo de la forma habitual (véase la parte I de este mismo capítulo) simplemente modificando los punteros adecuados. Se usa la bandera INDEXA para indicar si el puntero que señala el objeto está en la tabla de referencia o en otro elemento.

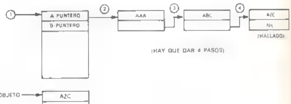
BUSQUEDA

El programa de búsqueda (BUSQ) reside en las posiciones de memoria 0108 ■ 015D y utiliza la subrutina PRETAB de la dirección 01D9.

El principio de búsqueda es bastante obvio:

1. Se obtiene en el directorio la entrada correspondiente a la letra del alfabeto situada en la primera posición de la etiqueta de objeto.
2. Se obtiene el puntero y se accede al elemento. Si es NIL, la entrada no existe.
3. En caso contrario, se compara el elemento con OBJETO; ■ coinciden, la búsqueda ha concluido positivamente. Si no, el apuntador se orienta hacia la entrada inmediatamente inferior.
4. Se vuelve a 2.

La figura 9.26 recoge un ejemplo.



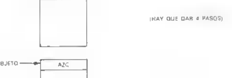


Figura 9.26
Búsqueda en ■ lista encadenada.

Figura 9.27
Ejemplo de inserción en la encadenada.

INSERCION

Es básicamente una búsqueda seguida de una inserción cuando se encuentra un elemento "NIL".

Se sitúa un bloque de almacenamiento, para la nueva incorporación, a continuación del marcador EOT, buscando para ello un indicador de ocupación en posición "disponible".

El programa aparece en la figura 9.29, se llama "NUEVO" y reside en las direcciones 015E a 01AB. La figura 9.27 recoge un ejemplo.

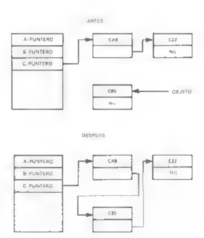
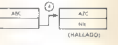


Figura 9.27
Ejemplo de inserción en la lista encadenada.

ELIMINACION

Para eliminar un elemento se sitúa su indicador de ocupación en posición "disponible" y se ajusta el puntero al mismo desde el directorio o desde el elemento anterior.

El programa se llama "BORRAR", y reside en las direcciones 01AC a 01D8. La figura 9.28 recoge un ejemplo de eliminación.



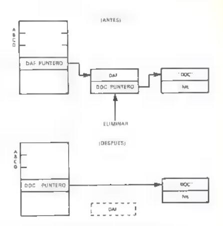


Figura 9.28
Ejemplo de eliminación.

Figura 9.29
Programas de la lista enca
nada.

```
0100                         ORG  #0100
0100 EE01          INDEXA DEFW FIN
0102 EF01          BASETA DEFW FIN+1
0104 F101          REFBAS DEFW FIN+3
0106 F301          LONENT DEFW FIN+5
                   ;
0108 3E00          BUSQ   LD   A,0            ;INICIALIZA FLAGS
010A 47                   LD   B,A
010B 3C                   INC  ■
010C 320001               LD   (INDEXA),A
010F CDD901               CALL PRETAB         ;COGE LA DIRECCION DEL
                   ;                          ;PUNTERO DE INDICE
0112 1A                   LD   A,(DE)         ;COLOCA EL CONTENIDO DEL
                   ;                          ;PUNTERO EN HL
0113 6F                   LD   L,A
0114 13                   INC  DE
0115 1A                   LD   A,(DE)
0116 67                   LD   H,A
0117 E5                   PUSH HL
0118 DDE1                 POP  IX
011A DD7E00        COMPAR LD   A,(IX+0)       ;MIRA LA PRIMERA LETRA DE
                   ;                          ;ENTRADA
011D FE7C                 CP   #7C            ;VE SI ES MARCADOR EOT
011F D25D01               JP   NC,NOENCO
0122 DD7E00               LD   A,(IX+0)       ;COMPARA LAS PRIMERAS LETR.
0125 FDBE00               CP   (IY+0)
0128 DA4601               JP   C,NOVALE
012B C25D01               JP   NZ,NOENCO
012E DD7E01               LD   A,(IX+1)       ;COMPARA SEGUNDAS LETRAS
0131 FDBE01               CP   (IY+1)
0134 DA4601               JP   C,NOVALE
0137 C25D01               JP   NZ,NOENCO
013A DD7E02               LD   A,(IX+2)       ;COMPARA TERCERAS LETRAS
013D DDBE02               CP   (IY+2)
0140 CA5B01               JP   Z,ENCONT
0143 D25D01               JP   NC,NOENCO
0146 DDE5          NOVALE PUSH IX
0148 D1                   POP  DE
0149 2A0601               LD   HL,(LONENT)    ;SALTA AL PUNTERO DE ENTR.
014C 19                   ADD  HL,DE
014D 4E                   LD   C,(HL)         ;PONE EL VALOR DEL PUNTERO
                   ;                          ;EN BC
014E 23                   INC  HL
014F 46                   LD   B,(HL)
0150 C5                   PUSH BC             ;CARGA IX CON EL PUNTERO
0151 DDE1                 POP  IX
0153 3E00                 LD   A,0
0155 320001               LD   (INDEXA),A     ;RESET DE FLAG
0158 C31A01               JP   COMPAR
015B 06FF          ENCONT LD   B,#0FF
015D C9            NOENCO RET
                   ;
                   ;
                   ;
015E CD0801        NUEVO  CALL BUSQ           ;VE DONDE DEBE IR EL OBJETO
0161 04                   INC  B
0162 CAAB01               JP   Z,SALIR
0165 D5                   PUSH DE             ;ALMACENA DIRECCION PREVIA
                   ;                          ;DE ENTRADA
0166 2A0201               LD   HL,(BASETA)    ;BUSCA ESPACIO EN LA TABLA
                   ;                          ;PARA ENTRADAS NUEVAS
0169 EB            SIGUIE EX   DE,HL          ;MUEVE AL FINAL DE LA
                   ;                          ;ENTRADA SIGUIENTE
016A 2A0601               LD   HL,(LONENT)
016D 23                   INC  HL             ;SUMA TRES PARA LA LONGITUD
                   ;                          ;DE ENTRADA REAL
016E 23                   INC  HL
016F 23                   INC  HL
0170 19                   ADD  HL,DE
0171 7E                   LD   A,(HL)
0172 3D                   DEC  A
0173 CA6901               JP   Z,SIGUIE       ;SI HAY ALGO ALLI, LO
                   ;                          ;INTENTA OTRA VEZ
```

Figura 9.29
Programas de la lista encadenada.

```
0176 13                 INC  DE
0177 D5                 PUSH DE          ;GUARDA LA POSICION DEL
                ;                        ;ESPACIO VACIO
0178 FDE5               PUSH IY          ;MUEVE IY A HL
017A E1                 POP  HL
017B ED4B0601           LD   BC,(LONENT) ;COLOCA EL OBJETO [illegible] LA
                ;                        ;TABLA
017F EDB0               LDIR
0181 DDE5               PUSH IX          ;PONE LA DIRECCION DE
                ;                        ;ENTRADA DETRAS DEL OBJETO
0183 E1                 POP  HL          ;...EN LA POSICION DEL
                ;                        ;PUNTERO
0184 [illegible]        EX   DE,HL
0185 73                 LD   (HL),E
0186 23                 INC  HL
0187 72                 LD   (HL),D
0188 23                 INC  HL
0189 3601               LD   (HL),1      ;DEFINE EL MARCADOR DE
                ;                        ;OCUPACION
018B E1                 POP  HL          ;COGE LA DIRECCION DONDE
                ;                        ;ESTA EL ESPACIO
018C 3A0001             LD   A,(INDEXA)  ;VE QUE PUNTEROS PREVIOS
018F 3D                 DEC  A           ;DEBE ESTABLECER
0190 CAA001             JP   Z,SETINX
0193 E3                 EX   (SP),HL     ;COGE LA DIRECCION DE
0194 ED5B0601           LD   DE,(LONENT) ;ENTRADA FREVIA AL OBJETO
0198 19                 ADD  HL,DE       ;Y LA COLOCA EN EL AREA
                ;                        ;DEL PUNTERO
0199 D1                 POP  DE          ;RECOBRA LA DIRECCION DEL
                ;                        ;OBJETO
019A 73                 LD   (HL),E      ;LA PONE EN LA POSICION
                ;                        ;DEL PUNTERO
019B 23                 INC  HL
019C 72                 LD   (HL),D
019D C3AB01             JP   TERMIN
01A0 C1         SETINX  POP  BC          ;LIMPIA EL STACK
01A1 CDD901             CALL PRETAB      ;COGE LA DIRECCION DEL
                ;                        ;INDICE
01A4 EB                 EX   DE,HL       ;CARGA HL EN EL
01A5 73                 LD   (HL),E
01A6 23                 INC  HL
01A7 72                 LD   (HL),D
01A8 01FFFF     TERMIN  LD   BC,#0FFFF   ;MUESTRA QUE HA HECHO
01AB [illegible] SALIR  RET
                ;
                ;
                ;
01AC CD0801     BORRAR  CALL BUSQ        ;COGE LA DIRECCION DEL
                ;                        ;OBJETO
01AF [illegible]        INC  B           ;VE SI ESTA ALLI
01B0 C2D801             JP   NZ,SALIRE
01B3 DDE5               PUSH IX          ;FIJA HL AL AREA [illegible] PUNTERO
                ;                        ;DEL OBJETO
01B5 E1                 POP  HL
01B6 ED4B0601           LD   BC,(LONENT)
01BA 09                 ADD  HL,BC
01BB 4E                 LD   C,(HL)      ;RECOBRA EL PUNTERO
01BC 23                 INC  HL
01BD 46                 LD   B,(HL)
01BE 23                 INC  HL
01BF 3600               LD   (HL),0      ;BORRA EL MARCADOR DE OCUP.
01C1 3A0001             LD   A,(INDEXA)  ;VE SI ES NECESARIO CAMBIAR
                ;                        ;EL INDICE
01C4 3D                 DEC  A
01C5 C2CF01             JP   NZ,CAMBIA
01C8 CDD901             CALL PRETAB      ;PONE LA DIRECCION EN HL
01CB [illegible]        EX   DE,HL
01CC C3D301             JP   MOVIN
01CF 2A0601     CAMBIA  LD   HL,(LONENT) ;FIJA HL AL PUNTERO DE LA
                ;                        ;ENTRADA PREVIA
01D2 19                 ADD  HL,DE
01D3 71         MOVIN   LD   (HL),C      ;          INC  HL
01D4 70                 LD   (HL),B
```

Figura 9.29
Programas de la lista encadenada (continuación).

Figura 9.29
Programas de la lista encadena
da (continuación).

Figura 9.30
Pase de prueba del program
la lista encadenada.

```
;GUARDA LA POSICION DEL
;ESPACIO VACIO
;MUEVE IY A HL

;COLOCA EL OBJETO EN LA
;TABLA

;PONE LA DIRECCION DE
;ENTRADA DETRAS DEL OBJETO
;...EN LA POSICION DEL
;PUNTERO



;DEFINE EL MARCADOR DE
;OCUPACION
;COGE LA DIRECCION DONDE
;ESTA EL ESPACIO
;VE QUE PUNTEROS PREVIOS
;DEBE ESTABLECER

;COGE LA DIRECCION DE
;ENTRADA PREVIA AL OBJETO
;Y LA COLOCA EN EL AREA
;DEL PUNTERO
;RECOBRA LA DIRECCION DEL
;OBJETO
;LA PONE EN LA POSICION
;DEL PUNTERO


;LIMPIA EL STACK
;COGE LA DIRECCION DEL
;INDICE
;CARGA HL EN HL


;MUESTRA QUE HA HECHO


;COGE LA DIRECCION DEL
;OBJETO
;VE SI ESTA ALLI

;FIJA HL AL AREA DE PUNTERO
;DEL OBJETO


;RECOBRA EL PUNTERO


;BORRA EL MARCADOR DE OCUP.
;VE SI ES NECESARIO CAMBIAR
;EL INDICE


;PONE LA DIRECCION EN HL


;FIJA HL AL PUNTERO DE LA
;ENTRADA PREVIA

;         INC   HL
```

```
01D5 01FFFF            LD    BC,#0FFFF
01D8 C9         SALIRE RET
                ;
                ;
                ;
01D9 E5         PRETAB PUSH  HL
01DA FD7E00            LD    A,(IY+0)     ;COGE LA PRIMERA LETRA DEL
                ;                         ;OBJETO
01DD 3D                DEC   A            ;BORRA LA CABECERA ASCII
01DE D640              SUB   #40
01E0 CB27              SLA   A            ;MULTIPLICA POR 2
01E2 2A0401            LD    HL,(REFBAS)
01E5 85                ADD   A,L
01E6 6F                LD    L,A
01E7 D2EB01            JP    NC,FIJAR
01EA 24                INC   H
01EB EB         FIJAR  EX    DE,HL
01EC E1                POP   HL
01ED C9                RET
                ;
01EE            FIN    END

BASETA 0102    BORRAR 01AC      NOVALE 0146    NUEVO  015E
BUSQ   0108    CAMBIA 01CF      PRETAB 01D9    REFBAS 0104
COMPAR 011A    ENCONT 0158      SALIR  01AB    SALIRE 01D8
FIJAR  01EB    FIN    01EE      SETINX 01A0    SIGUIE 0169
INDEXA 0100    LONENT 0106      TERMIN 01AB
MOVIN  01D3    NOENCO 015D
```

Figura 9.29
Programas de la lista encadenada (continuación).

Objetos en memoria

Relación de objetos y sus posiciones en memoria

```
DM300
0300  53 4D 4E 31 31 31 31 31 31 31 31 31 31 00 00 00   SMN1111111111...
0310  44 41 44 32 32 32 32 32 32 32 32 32 32 00 00 00   DAD2222222222...
0320  4D 4F 4D 33 33 33 33 33 33 33 33 33 33 00 00 00   MOM3333333333...
0330  55 4E 43 34 34 34 34 34 34 34 34 34 34 00 00 00   UNC4444444444...
0340  41 4E 54 35 35 35 35 35 35 35 35 35 35 00 00 00   ANT5555555555...
0350  41 41 41 36 36 36 36 36 36 36 36 36 36 00 00 00   AAA6666666666...
0360  41 56 50 37 37 37 37 37 37 37 37 37 37 00 00 00   AVP7777777777...
0370  53 48 44 38 38 38 38 38 38 38 38 38 38 00 00 00   SHD8888888888...
```

Carácter EOT en la tabla inicial

```
DM400
0400  [illegible] 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0410  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0420  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0430  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0440  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0450  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0460  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0470  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
```

Directorio inicial

```
DM500
0500  00 04 00 04 00 04 00 04 00 04 00 04 00 04 00 04   ................
0510  00 04 00 04 00 04 00 04 00 04 00 04 00 04 00 04   ................
0520  00 04 00 04 00 04 00 04 00 04 00 04 00 04 00 04   ................
0530  00 04 00 04 00 00 00 00 00 00 00 00 00 00 00 00   ................
0540  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0550  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0560  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0570  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
```

Figura 9.30
Pase de prueba del programa de la lista encadenada.

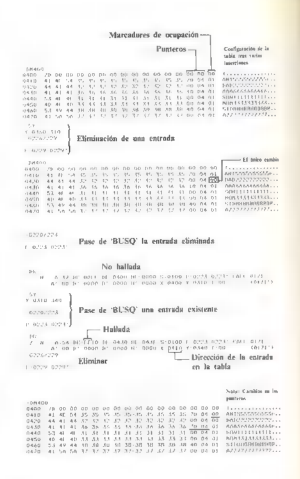


Figura 9.30
Pase de prueba del programa de la lista encadenada (continuación).

Resumen

El programador principiante no tiene necesidad de preocuparse todavía por los detalles de la realización y manipulación de estructuras de datos, aunque la programación eficiente de algoritmos no triviales obliga a dominarlas. Los ejemplos presentados en este capítulo ayudarán al lector a conseguir ese dominio y a resolver todos los problemas planteados por las estructuras de datos habituales.

ión

Configuración de la tabla tras varias inserciones

El único cambio

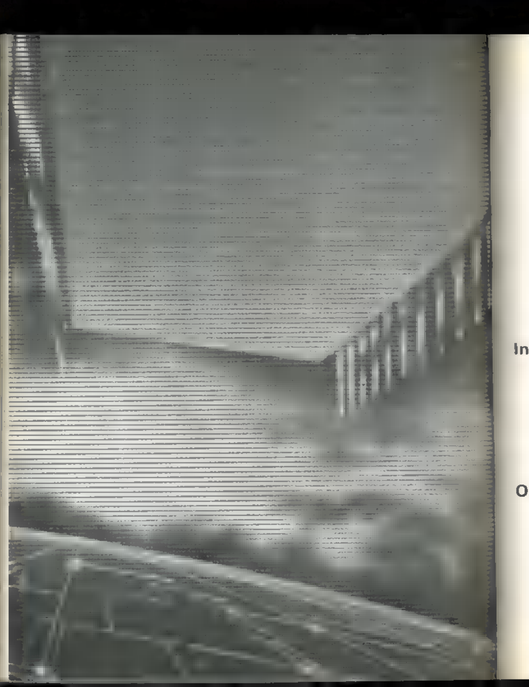
inada

istente

irección de la entrada
la tabla

Nota: Cambios en los punteros

necesidad de preocu-
zación y manipulación
gramación eficiente de
as. Los ejemplos pre-
ector a conseguir ese
as planteados por las



d

Introducción

Opciones bás

10
Desarrollo de programas

Introducción

Todos los programas que hemos visto hasta ahora los hemos desarrollado "a mano", sin ayuda del soporte lógico ni del físico. La única mejora sobre el uso directo del código binario ha sido la escritura en un código mnemotécnico llamado lenguaje ensamblador. Pero para desarrollar buenos programas es necesario conocer las posibilidades que ofrecen los soportes físico y lógico, y valorarlas es justamente la finalidad de este capítulo.

Opciones básicas de programación

Un programa puede escribirse fundamentalmente de tres formas: en código binario o hexadecimal, en lenguaje ensamblador o en un lenguaje de alto nivel. Analicemos una por una las tres posibilidades.

CODIFICACION HEXADECIMAL

Lo normal es escribir los programas en ensamblador, pero la mayor parte de los sistemas baratos de placa única no disponen del programa ensamblador encargado de traducir au-

tomáticamente los términos mnemotécnicos a código máquina, lo que obliga a efectuar esa traducción a mano. La codificación binaria es *muy molesta* de usar y está expuesta a errores, por lo que normalmente se prefiere la hexadecimal. Como ya vimos en el capítulo 1, una cifra hexadecimal representa cuatro bits binarios, de manera que el contenido de cualquier byte se representa con sólo dos. En el apéndice se encuentran los equivalentes hexadecimales de las instrucciones del Z80.

En resumen, si los recursos del usuario son limitados y no dispone de ensamblador, debe traducir a mano el programa a hexadecimal. Es una labor razonable si las instrucciones son pocas —entre 10 y 100, por ejemplo—, pero aburrida y propensa a errores si el programa es más largo; sin embargo, el hecho es que la mayor parte de los microordenadores de placa única obligan a trabajar en hexadecimal porque, para que salgan más baratos, se fabrican sin programa ensamblador y sin teclado alfanumérico.

La codificación hexadecimal no es, pues, la más aconsejable, sino la más barata. El precio de un ensamblador y de un teclado alfanumérico se compensa ampliamente con el trabajo que se ahorra al introducir el programa en memoria. En cualquier caso, ello no altera la forma en que se escribe el programa propiamente dicho: *esto se hace siempre en lenguaje ensamblador*, para que sea comprensible por el programador humano.

Figura 10.1
Niveles de programación.

PROGRAMACION EN LENGUAJE ENSAMBLADOR

Esta forma de programación se refiere tanto a los programas que se cargan en código hexadecimal como a los que se introducen en código simbólico mnemotécnico. Vamos a centrarnos en este segundo caso, que exige disponer del correspondiente programa ensamblador. Este lee cada una de las instrucciones simbólicas y las traduce al código binario utilizando de 1 a 5 bytes, según lo especificado al codificar las instrucciones. Pero un buen ensamblador dispone de otros recursos adicionales que facilitan la escritura de programas y que examinaremos más adelante. En particular, dispone de *seudoinstrucciones* que modifican el valor de los símbolos. Puede trabajarse con direccionamiento simbólico y saltarse a una posición simbólica. Durante la fase de puesta a punto, que suele suponer la eliminación o la adición de instrucciones, no es preciso reescribir el programa completo si se inserta alguna entre una bifurcación y el punto al que se bifurca, porque se emplean etiquetas simbólicas que el ensamblador ajustará automáticamente durante la traducción. Gracias al ensamblador, el programa puede también

ponerse a punto en forma simbólica. Para examinar el contenido de una posición de memoria y reconstruir la instrucción que representa a nivel ensamblador puede emplearse un desmontador o desensamblador. Más adelante veremos los recursos que ofrece el soporte lógico de un sistema, pero antes vamos a detenernos en la tercera opción.

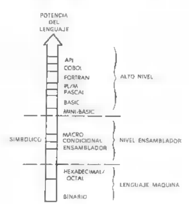


Figura 10.1
Niveles de programación.

LENGUAJES DE ALTO NIVEL

Los programas pueden, por último, escribirse en BASIC, APL, PASCAL u otro cualquiera de los numerosos lenguajes de alto nivel. Las técnicas de trabajo con éstos están fuera del alcance de este libro, y nos limitaremos aquí a reseñar brevemente algunas peculiaridades de las mismas. Los lenguajes de alto nivel constan de instrucciones potentes que hacen la labor de programación mucho más rápida y mucho más fácil. Dentro del ordenador hay un programa muy complejo que traduce dichas instrucciones a la representación binaria de máquina. Por lo general, a cada instrucción de alto nivel corresponden muchas binarias. El programa encargado de la traducción se llama *compilador* o *intérprete*. El compilador traduce primero el programa entero a código objeto y a continuación lo ejecuta; el intérprete, por el contrario, traduce una instrucción y la ejecuta antes de pasar a la siguiente; tiene la ventaja de la interactividad, pero a cambio de una eficacia inferior a la del compilador. No diremos nada más aquí de esta clase de lenguajes, y nos limitaremos a partir de ahora a la programación de un microprocesador real en lenguaje ensamblador.

Recursos lógicos

Veremos aquí los recursos lógicos más importantes que hay, o debe haber, en los sistemas de desarrollo completos. Ya hemos definido algunos de ellos; antes de seguir los describiremos brevemente y definiremos los demás programas de interés.

El *ensamblador* es el programa encargado de traducir la representación mnemotécnica de las instrucciones a su equivalente binario. A cada instrucción simbólica suele corresponder una binaria (que puede ocupar 1, 2 ó 3 bytes). El código binario resultante se llama *código objeto* y es directamente ejecutable por el microordenador. El ensamblador genera también un listado simbólico completo del programa, las tablas de equivalencia que debe usar el programador y la lista de frecuencia de aparición de símbolos en el programa. Veremos algunos ejemplos en este mismo capítulo.

También hace el ensamblador una relación de errores de sintaxis, como instrucciones mal escritas o ilegales, fallos de bifurcación, etiquetas duplicadas o inexistentes, etc.

Pero no elimina los errores *lógicos*, que son competencia exclusiva del *programador*.

El programa *compilador* traduce las instrucciones escritas en lenguaje de alto nivel a su forma binaria.

El *intérprete* es parecido al compilador, ya que también traduce instrucciones de alto nivel a representación binaria, pero no conserva la representación intermedia de las mismas y, además, las ejecuta inmediatamente. Incluso es normal que ni siquiera genere código intermedio y ejecute directamente las instrucciones de alto nivel.

El *monitor* es un programa básico indispensable para utilizar los recursos físicos del sistema. Controla constantemente las entradas de los periféricos y organiza el resto de los dispositivos. Por ejemplo, en un microordenador de placa única equipado con teclado e indicadores LED, el monitor mínimo, debe vigilar constantemente el teclado por si se produce una entrada y presentar el contenido especificado en la pantalla LED; también debe comprender unas pocas órdenes introducidas por teclado, como ARRANCAR, PARAR, SEGUIR, CARGAR EN MEMORIA o EXAMINAR MEMORIA. En sistemas grandes suele llamarse al monitor programa *ejecutivo*, ya que también se encarga de efectuar manipulaciones complejas en los ficheros y de organizar las tareas. Todo este conjunto de recursos constituye el *sistema operativo*; cuando los ficheros residen en disco, el sistema operativo se denomina *sistema operativo de disco* o DOS *(disk operative system)*.

La finalidad del *editor* es facilitar la introducción y modificación de textos y programas. Permite al usuario introducir caracteres cómodamente, suplementarlos, insertarlos, añadir líneas, eliminar líneas y buscar caracteres o series de caracteres. Es un recurso importante que facilita la introducción de textos.

Cuando un programa no funciona bien, lo normal es que no haya ninguna indicación del origen del fallo, y por eso el programador suele insertar puntos de interrupción que lo detienen en direcciones especificadas para poder examinar el contenido de la memoria en esos puntos; ésta es, justamente, la función principal del programa de *puesta a punto (debugger)*. Permite interrumpir un programa, volver a ponerlo en marcha, examinarlo y visualizar y modificar el contenido de los registros de memoria. Los buenos programas de puesta a punto también analizan datos en forma simbólica, hexadecimal, binaria u otra representación habitual, y los cargan en esos mismos formatos.

El *programa de carga* es responsable de la colocación de varios bloques de código objeto en posiciones especificadas de memoria y del ajuste de sus respectivos apuntadores simbólicos para que puedan referenciarse mutuamente. Se emplea para desplazar programas o bloques a diversas áreas de memoria. El programa *simulador* o *emulador* imita el funcionamiento de un dispositivo, por lo general el microprocesador, en su ausencia cuando se desarrolla un programa sobre un procesador simulado antes de cargarlo en la placa real. De esta forma se puede suspender un programa, modificarlo y archivarlo en memoria RAM; pero el simulador también tiene inconvenientes:

1. Por lo general, sólo simula el procesador, no los dispositivos de entrada/salida.
2. La velocidad de ejecución es baja y opera en tiempo simulado, lo que impide la verificación de dispositivos de tiempo real y puede plantear problemas de sincronización aun cuando la lógica del programa sea correcta.

Un *emulador* es, básicamente, un simulador de tiempo real. Utiliza un procesador para simular otro con todo detalle.

Se llaman *rutinas de servicio* a las necesarias en la mayor parte de las aplicaciones y de las que el usuario quisiera ver encargarse al fabricante: multiplicación, división y otras operaciones aritméticas, desplazamiento de bloques, verificación de caracteres, manipulación de dispositivos de entrada/salida, etc.

Secuencia de desarrollo de un programa

Vamos a analizar la secuencia típica de desarrollo de un programa a nivel de ensamblador. Supondremos que disponemos de todos los recursos mencionados antes, con el fin de poner de relieve su utilidad. Cuando no se encuentran en un sistema, siempre pueden reemplazarse por los correspondientes programas, pero a costa de un mayor esfuerzo de puesta a punto.

Lo normal es empezar por crear un algoritmo y definir las estructuras de datos del problema que desea resolverse. A continuación hay que desarrollar todos los diagramas de flujo necesarios; y, por último, convertir éstos en un programa escrito en lenguaje ensamblador (esta última fase se llama codificación).

Acto seguido se introduce el programa en el ordenador, para lo que existen diversos recursos físicos que estudiaremos en la siguiente sección.

El programa pasa a la memoria RAM del sistema bajo la supervisión del editor. Cuando ha entrado una parte completa del mismo —una o varias subrutinas—, se comprueba su funcionamiento.

Primero se utiliza el programa ensamblador. Si no reside en el sistema, se carga a partir de una memoria externa, como un disco. A continuación el programa se ensambla, es decir, se traduce a código binario, lo que da lugar al programa objeto, listo ya para ser ejecutado.

Es raro que un programa funcione bien a la primera. Para verificar su buen comportamiento se introducen en posiciones cruciales, en las que sea fácil comprobar si los resultados intermedios son correctos, una serie de puntos de interrupción. Para efectuar la comprobación se emplea el programa de puesta a punto. Al dar la orden "ARRANQUE", el programa se pone en marcha y se detiene en los puntos especificados, que el programador aprovecha para comprobar el contenido de los registros, o de la memoria, y asegurarse de que los datos son correctos; en caso afirmativo, sigue hasta el punto siguiente. Si aparece un dato incorrecto, es que hay un error en el programa; lo primero que hace en este caso el programador es repasar el listado para ver si los códigos están bien escritos; si lo están, cabe suponer que el error es de naturaleza lógica, y en ese caso conviene estudiar los diagramas de flujo (estamos suponiendo que éstos se han comprobado previamente a mano y funcionan razonablemente bien). Lo normal es que el fallo esté en la codificación, lo que obliga a rehacer un segmento del programa. Si la representación simbólica del mismo está todavía en memoria, no hay más que reintroducir el editor y modificar las líneas afectadas.

para a continuación repetir de nuevo la sección corregida. En algunos sistemas la memoria no es suficientemente capaz, y es preciso llevar la representación simbólica del programa a un disco o una cinta antes de ejecutar el código objeto; en este caso, como es natural, hay que volver a cargar la representación simbólica a partir del soporte antes de introducir de nuevo el editor.

El procedimiento descrito se repite todas las veces que sea necesario hasta que el programa funcione correctamente. Es preciso insistir en que la prevención es mucho más eficaz que la curación. Si el diseño es correcto, el programa empezará a funcionar bien rápidamente en cuanto se corrijan los errores mecanográficos inevitables o los fallos de codificación obvios; por el contrario, poner a punto un programa mal diseñado lleva muchísimo tiempo, más todavía que el empleado en el diseño. Por tanto, vale más dedicar más tiempo a éste y reducir, en cambio, el de corrección.

Pero, aunque de esta forma se pone a prueba la organización general del programa, no se verifica el funcionamiento en tiempo real con dispositivos de entrada/salida. Si hay que hacer esta verificación, lo más inmediato es cargar el programa en EPROM, instalarlo en la placa y observar si funciona.

Pero hay una solución mejor: consiste en utilizar un *circuito emulador interno* que utiliza el microprocesador Z80 (o cualquier otro) para emular al Z80 casi en tiempo real. La emulación es física; el dispositivo está equipado con una línea acabada en un conector de 40 patillas, dispuestas exactamente igual que las del Z80, que se acopla a la placa real de la aplicación en la que se está trabajando. Las señales generadas por el emulador son idénticas a las del Z80, aunque quizá un poco más lentas. La ventaja más importante es que el programa en estudio sigue residiendo en la RAM del sistema de desarrollo y genera las señales reales de comunicación con los dispositivos reales de entrada/salida que quieran utilizarse. De esta manera puede trabajarse en el programa con todos los recursos del mencionado sistema de desarrollo (edición, puesta a punto, recursos simbólicos, fichero) y a la vez comprobar la entrada/salida en tiempo real.

Pero un buen emulador no se limita a esto, sino que ofrece, además, otras posibilidades, como el *analizador*, que registra las últimas instrucciones o el estado de los diversos *buses* del sistema antes de un punto de interrupción; en otras palabras, el analizador hace un resumen de los acontecimientos ocurridos antes del punto de interrupción o del error; incluso puede examinar una dirección determinada o la aparición de una combinación de bits específica. Es un recurso muy valioso,

porque cuando se encuentra un error suele ser demasiado tarde, puesto que la instrucción o el dato que lo han provocado son anteriores a la detección. El analizador permite al programador localizar el segmento causante del error; si el segmento analizado no fuese suficientemente largo, bastaría situar antes el punto de interrupción.

Con esto termina la descripción de la serie de operaciones de que consta el desarrollo de un programa. Pasemos ahora a describir los recursos físicos que facilitan dicho desarrollo.

ROM	RAM
INICIALIZACION	ENSAMBLADOR O COMPILADOR O INTERPRETE
MANDO DEL TECLADO	DOS
MANDO DE PANTALLA	EDITOR O PUESTA A PUNTO O SIMULADOR
MANDO TELETIPO	ESPACIO DEL SISTEMA (Y LA PILA)
MANDO CASSETTE	PROGRAMA DEL USUARIO
INTERPRETE DE ORDENES	ESPACIO DEL USUARIO
RUTINAS DE SERVICIO	
PUESTA A PUNTO ELEMENTAL	
EDITOR ELEMENTAL	

Figura 10.2
Un mapa de memoria típico.

le ser demasiado tarde.
lo han provocado son
ermite al programador
si el segmento analiza-
a situar antes el punto

a serie de operaciones
ma. Pasemos ahora a
n dicho desarrollo.

Recursos físicos

MICROORDENADOR MONOPLACA

El microordenador de una sola placa es la forma más barata de iniciarse en el desarrollo de programas. Normalmente dispone de un teclado hexadecimal, unas pocas teclas de funciones y 6 indicadores LED que visualizan direcciones y datos. Dado que la memoria es reducida, el aparato suele carecer de ensamblador; en el mejor de los casos tiene un pequeño monitor y apenas recursos de edición y puesta a punto, con excepción de unas pocas órdenes; por tanto, todos los programas deben introducirse en forma hexadecimal, y así es también como aparecen en los indicadores LED. En teoría, un microordenador monoplaca tiene la misma potencia física que cualquier otro, y si no soporta los recursos habituales de un sistema mayor y alarga mucho el desarrollo de programas es únicamente por lo reducido de su memoria. Como trabajar en sistema hexadecimal es muy pesado, los microordenadores monoplaca son adecuados, sobre todo, para estudiar programas educativos y de adiestramiento, habitualmente bastante cortos. Constituyen, sin duda, la forma más económica de aprender a programar practicando, pero para desarrollar con ellos aplicaciones complejas es imprescindible conectarles pastillas de memoria adicionales capaces de albergar los recursos lógicos habituales.

SISTEMAS DE DESARROLLO

Un sistema de desarrollo es un microordenador con una cantidad considerable de memoria RAM (32K o 48K) y los dispositivos de entrada/salida adecuados: pantalla, impresora, discos y, habitualmente, un programador de PROM, además de un circuito emulador. Es un aparato creado específicamente para facilitar el desarrollo de programas en el medio industrial. Normalmente ofrece todos o casi todos los recursos lógicos mencionados en la sección anterior, por lo que, en principio, constituye el instrumento de desarrollo de programas idóneo.

Tiene el inconveniente de que, por lo general, no soporta compilador ni intérprete, que normalmente ocupan mucha memoria, más de la que posee el sistema. De todas formas, ofrece todo lo necesario para crear programas en lenguaje ensamblador. Lo malo es que, como se vende mucho menos que los ordenadores para aficionados, cuesta bastante más caro.

MICROORDENADORES PARA AFICIONADOS

El soporte físico de uno de estos microordenadores es exactamente igual al de un sistema de desarrollo. La diferencia principal radica en que normalmente no disponen de los refinados recursos lógicos instalados en los aparatos industriales. Así, cuentan con ensambladores elementales, y editores y sistemas de ficheros mínimos, y carecen de programador de PROM, de circuito emulador interno y de sistemas potentes de puesta a punto. Representan, pues, una opción intermedia entre un microordenador monoplaca y un sistema de desarrollo. Para quien desee crear programas de complejidad modesta constituyen, sin duda, el mejor compromiso entre la economía de adquisición y una cantidad aceptable de instrumentos de desarrollo.

SISTEMAS EN TIEMPO COMPARTIDO

Varias firmas alquilan terminales conectados a redes de tiempo compartido que utilizan grandes ordenadores y aprovechan sus ventajas. Casi todos estos sistemas disponen de *ensambladores cruzados* para todos los microordenadores. Un ensamblador cruzado no es más que un ensamblador para el procesador X que reside en el procesador Y (por ejemplo: un ensamblador para el Z80 instalado en un IBM 370). El tipo del ordenador central carece de importancia. El usuario escribe sus programas en el ensamblador del Z80, y el ensamblador cruzado los traduce al código binario apropiado. La diferencia estriba en que el programa no puede ejecutarse más que en un procesador simulado y siempre que no utilice recursos de entrada/salida. Es, pues, una solución limitada al medio industrial.

ORDENADOR GRANDE

Si se tiene acceso a un ordenador grande y potente, también puede emplearse un ensamblador cruzado. Cuando el ordenador está disponible en tiempo compartido, la opción es básicamente igual a la anterior. Pero el servicio por lotes constituye uno de los medios de desarrollo de programas más incómodo, porque el tiempo de trabajo se alarga muchísimo.

¿CON PANEL FRONTAL O SIN PANEL FRONTAL?

El panel frontal es un accesorio del soporte físico que suele utilizarse en la puesta a punto de programas. Tradicionalmente para visualizar cómodamente el contenido binario de un registro o de la memoria. Sin embargo, todas sus funciones puede ejecutarlas un terminal, y la actual pantalla de rayos catódicos ofrece un servicio similar al del panel, con la ventaja de que en ella es fácil pasar de representación binaria a hexadecimal, simbólica o decimal (siempre que se disponga de las correspondientes rutinas de conversión, por supuesto). El inconveniente es que para obtener la visualización deseada hay que pulsar varias teclas en lugar de girar un botón. De todas formas, como el panel es bastante caro, la mayor parte de los microordenadores actuales han prescindido de este dispositivo de puesta a punto. Su valor se defiende, con frecuencia, más con argumentos sentimentales, justificados por la experiencia pasada del programador, que con razones reales. Desde luego, no es imprescindible.

RESUMEN DE RECURSOS FISICOS

A grandes rasgos, cabe distinguir tres situaciones: Quien dispone de un presupuesto mínimo y desea aprender a programar deberá adquirir un microordenador monoplaca que le permita desarrollar todos los programas sencillos de este libro y muchos más. Sus limitaciones se hacen sentir en cuanto los programas superan unos pocos cientos de instrucciones.

El usuario industrial necesitará un sistema de desarrollo, ya que cualquier otro medio más pobre en recursos alargará considerablemente el tiempo de trabajo necesario para poner a punto los programas. La equivalencia está clara: a mejor soporte físico, menos tiempo de programación. Naturalmente, para desarrollar programas sencillos puede seguirse un procedimiento más barato, pero si los programas son complejos no tiene sentido escatimar en la adquisición de recursos físicos, porque la parte más costosa del desarrollo será justamente la programación.

Para el aficionado bastarán los recursos mínimos, pero suficientes, que proporcionan los microordenadores domésticos. Todavía faltan por desarrollar muchos recursos lógicos de desarrollo para tales microordenadores, y el usuario deberá evaluar su sistema a la vista de las observaciones hechas en este capítulo.

Pasemos ya a analizar con detalle el recurso más importante de todos: el programa ensamblador.

El programa ensamblador

A lo largo de todo el libro hemos estado utilizando el lenguaje ensamblador sin hablar de la síntesis formal del mismo ni definirlo. La finalidad de este lenguaje es proporcionar al usuario una representación simbólica fácil de usar y que, a la vez, pueda traducir rápidamente el correspondiente programa ensamblador a la notación binaria.

CAMPOS DEL ENSAMBLADOR

He aquí los campos utilizados al escribir un programa en lenguaje ensamblador:

— el *campo de etiqueta*, opcional, que puede contener una dirección simbólica para la instrucción que sigue;
— el *campo de instrucción*, que contiene el código de operación y los operandos (puede establecerse un campo de operandos independiente);
— el *campo de comentarios*, situado a la derecha, opcional y encaminado a facilitar la lectura del programa.

Los tres aparecen en el formulario de programación de la figura 10.3.

Cuando el ensamblador recibe el programa, confecciona un *listado* del mismo, operación que conlleva la creación de tres nuevos campos, situados, por lo general, a la izquierda de la página (véase el ejemplo de la figura 10.4). Cada una de las líneas escritas por el programador recibe un número simbólico que se escribe a la izquierda del todo.

El siguiente campo avanzando hacia la derecha es el campo de la dirección real, que recoge en formato hexadecimal el valor del contador del programa que señala la instrucción de que se trate.

A su derecha se encuentra la representación hexadecimal de la instrucción.

Esto sugiere una de las posibles aplicaciones del programa ensamblador, aunque, cuando se trabaje con un microordenador monoplaca que sólo acepte el código hexadecimal, puede pasarse el programa escrito en ensamblador por un sistema que disponga del correspondiente programa, si se tiene acceso al mismo, que generará la codificación hexadecimal correcta.

Figura 10.3
Formulario de programaci
un microprocesador.

DIRECCIÓN	INSTRUCCION HEXADECIMAL 1	2	3	4	ETIQUETA	CODIGO OPERATIVO SIMBOLICO	OPERANDO	OBSERVACIONES

Figura 10.3
Formulario de programación de un microprocesador.

TABLAS

Cuando traduce el programa simbólico [redacted] notación binaria, el ensamblador realiza dos tareas esenciales:

1. Pasa las instrucciones mnemotécnicas [redacted] código binario.
2. Pasa los símbolos que representan constantes y direcciones a notación binaria.

Para facilitar la puesta a punto del programa, el ensamblador presenta al final del listado la equivalencia entre los símbolos y su valor hexadecimal; es lo que se llama la tabla de símbolos.

Algunas tablas no se limitan a recoger el símbolo y [redacted] valor, sino que, además, indican los números de línea en que aparecen dichos símbolos.

MENSAJES DE ERROR

Durante la traducción, el ensamblador detecta los errores de sintaxis y los incluye en el listado final. Son diagnósticos habituales los siguientes: símbolos sin definir, etiquetas ya definidas, código de operación, direcciones o modos de direccionamiento incorrectos. Naturalmente, es deseable disponer de diagnósticos más detallados, como los que proporcionan algunos programas ensambladores.

EL LENGUAJE ENSAMBLADOR

Ya hemos definido los códigos de operación, así que describiremos aquí los símbolos, las constantes y los operadores que pueden utilizarse como parte de la sintaxis del ensamblador.

SIMBOLOS

Sirven para representar valores numéricos, sean datos o direcciones. Pueden estar formados por hasta seis caracteres, y deben comenzar por uno alfabético; los otros cinco sólo pueden ser letras del abecedario [redacted] números; además, no pueden utilizarse los nombres de los códigos de operación, los de los registros (A, B, C, D, E, H, L, BC, DE, HL, AF, BC, DE, IX, IY y SP) ni los de los seudooperadores utilizados por el programa ensamblador (los nombres de estas seudooperaciones aparecen más adelante, en la sección correspondiente). Tampoco está permitida la utilización como símbolos de las denominaciones de las banderas: C, Z, N, PE, NC, P, PO, NZ y M.

Figura 10.4
Ejemplo de salida del ensamb
dor.

Asignación de un valor a un símbolo

Las etiquetas son símbolos especiales cuyo valor no tiene que definir el programador, ya que será automáticamente asignado por el programa ensamblador conforme las vaya encontrando. De esta forma, el valor de la etiqueta corresponde automáticamente a la dirección de la instrucción generada en la línea en que se encuentra. Hay seudoinstrucciones especiales para forzar nuevos valores de partida a las etiquetas o para asignarles valores específicos.

```
0100              10           ORG   #0100
0100 0002         20 MPRAD     DEFW  #0200
0102 0202         30 MPDAD     DEFW  #0202
0104 0402         40 RESAD     DEFW  #0204
                  50 ;
0106 ED4B0001     60 MP488     LD    BC,(MPRAD)     ;CARGA MULTIPLICADOR EN C
010A 0608         70           LD    B,8            ;B ES CONTADOR DE BIT
010C ED5B0201     80           LD    DE,(MPDAD)     ;CARGA MULTIPLICANDO EN E
0110 1600         90           LD    D,0            ;INICIALIZA D
0112 210000      100           LD    HL,0           ;PONE A 0 EL RESULTADO
0115 CB39        110 MULT      SRL   C              ;SHIFT AL ACARREO DEL BIT
                 115 ;                              ;MULTIPLICADOR
0117 3001        120           JR    NC,NOADD       ;COMPRUEBA EL ACAREO
0119 19          130           ADD   HL,DE          ;SUMA AL RESULTADO MPD
011A CB23        140 NOADD     SLA   E              ;SHIFT IZQUIERDA DE MPD
011C CB12        150           RL    D              ;GUARDA EL BIT EN D
011E 05          160           DEC   B              ;DECR EL CONT DE SHIFT
011F C21501      170           JP    NZ,MULT        ;REPETIR SI CONTADOR<>0
0122 220401      180           LD    (RESAD),HL     ;ALMACENA EL RESULTADO
                 190 END
```

Figura 10.4
Ejemplo de salida del ensamblador.

Por el contrario, el programador debe definir, antes de usarlos, los valores asignados a los símbolos correspondientes a constantes y direcciones de memoria.

La asignación de valor se realiza por medio de seudoinstrucciones, que son instrucciones para el ensamblador que no se traducen en otras ejecutables. Así, la constante LOG se definiría:

```
LOG    EQU    3002H
```

la sentencia asigna el valor hexadecimal 3002 a la variable LOG. En una próxima sección estudiaremos las seudoinstrucciones del ensamblador.

CONSTANTES O LITERALES

Tradicionalmente, las constantes pueden expresarse en notación decimal, hexadecimal, octal o binaria o como series de caracteres alfanuméricos. La base de representación empleada se

indica mediante un símbolo. Para cargar "0" en el acumulador basta escribir

```
LD    A, 0
```

aunque opcionalmente puede añadirse una "D" al final de la constante.

Los números hexadecimales terminan con el símbolo "H". Para cargar el valor "FF" en el acumulador, se escribe:

```
LD    A, 0FFH
```

Los valores octales terminan con los símbolos "O" o "Q", y los binarios, con el "B".

Por ejemplo, para cargar en el acumulador el valor "11111111", se escribe:

```
LD    A, 11111111B
```

En el campo literal son también admisibles los caracteres literales ASCII, que deben ir encerrados entre comillas simples. Así, para cargar el símbolo "S" en el acumulador se escribe:

```
LD    A, 'S'
```

Ejercicio 10.1: *¿Cargarán el mismo valor en el acumulador las dos instrucciones LD A, '5' y LD A, 5H?*

Obsérvese que en la convención de Zilog los paréntesis denotan direcciones. Por ejemplo:

```
LD    A, (10)
```

especifica que el acumulador se carga con el contenido de la dirección decimal de memoria (10).

OPERADORES

Los operadores son recursos del ensamblador que facilitan la escritura de programas simbólicos. Hacen falta como mínimo los signos más y menos para poder especificar, por ejemplo:

```
LD    A, (DIRECCION)
LD    A, (DIRECCION + 1)
```

Es importante darse cuenta de que la expresión DIRECCION + 1 será utilizada por el ensamblador para determinar la dirección real de memoria que debe utilizar como equivalente binario; es decir, la expresión se calcula durante la *fase de ensamblado*, no durante la ejecución del programa.

Puede haber otros operadores, como la multiplicación y la división, útiles para acceder a tablas de memoria. O algunos más especializados, como mayor que y menor que, que truncan valores de dos bytes en sus componentes inferior y superior.

Naturalmente, las expresiones deben dar lugar a valores positivos. Normalmente no se emplean números negativos, que, en todo caso, deben representarse en formato hexadecimal.

Por último, se emplea el símbolo especial "$" para representar el valor en curso de la dirección de la línea; debe interpretarse como "posición actual" (valor del PC).

Ejercicio 10.2: *¿Qué diferencia hay entre las dos instrucciones siguientes?:*

```
LD    A,10101010B
LD    A,(10101010B)
```

Ejercicio 10.3: *¿Cuál será el resultado de la siguiente instrucción?:*

```
JR    NC,$ - 2
```

EXPRESIONES

El ensamblador del Z80 permite utilizar gran diversidad de expresiones con operaciones aritméticas y lógicas. Para calcularlas, avanza de izquierda a derecha, con arreglo a las prioridades que recoge la tabla de la figura 10.5. Para forzar un orden de evaluación determinado, puede utilizarse paréntesis, pero siempre teniendo en cuenta que los más externos indicarán que lo contenido entre ellos deberá tratarse como una dirección.

SEUDOINSTRUCCIONES DEL ENSAMBLADOR

Se llama así a las órdenes especiales que da el programador al ensamblador y que determinan el almacenamiento de valores en símbolos, o en memoria, o el control de la ejecución del programa, o de su impresión. Las que controlan precisamente la impresión se llaman también "órdenes", y las describiremos en una sección aparte.

Veamos ahora las 11 seudoinstrucciones con que cuenta el sistema de desarrollo Zilog:

ORG nn

iguala el contador de direcciones del ensamblador al valor nn; en otras palabras; la primera instrucción ejecutable que aparezca tras esta seudoinstrucción residirá en el valor nn; sirve para alojar diferentes segmentos del programa en posiciones de memoria específicas.

EQU nn

asigna un valor a una etiqueta.

DEFL nn

también asigna un valor nn a una etiqueta, pero puede repetirse dentro del programa con diferentes valores para la misma, al contrario que EQU, que sólo puede usarse una vez.

DEFB n

asigna un contenido de 8 bits al byte que reside en el contador de referencia en curso.

OPERADOR	FUNCION	PRIORIDAD
+	MAS UNARIO	1
–	MENOS UNARIO	1
.NOT. or \	NO LOGICO	1
.RES.	RESULTADO	1
**	POTENCIACION	2
*	MULTIPLICACION	3
/	DIVISION	3
.MOD.	MODULO	3
.SHR.	DESPLAZAMIENTO LOGICO DCHA.	3
.SHL.	DESPLAZAMIENTO A LA IZQ.	3
+	SUMA	4
-	RESTA	4
.AND. or &	Y LOGICO	5
.OR. or !	O LOGICO	6
.XOR.	O EXCLUSIVO LOGICO	6
.EQ. or =	IGUAL	7
.GT. or >	MAYOR QUE	7
.LT. or <	MENOR QUE	7
.UGT.	MAYOR QUE SIN SIGNO	7
.ULT.	MENOR QUE SIN SIGNO	7

Figura 10.5
Tabla de prioridad de los operadores.

DEFB 'S'

asigna el valor ASCII de "S" al byte

DEFW nn

asigna el valor nn a la palabra de dos bytes que reside en el contador de referencia en curso y en la posición siguiente.

DEFS nn

reserva un bloque de memoria de nn bytes de tamaño a partir del valor en curso del contador de referencia.

DEFM 'S'

almacena en memoria la serie 'S', que empieza en el contador de referencia en curso; debe tener una longitud inferior a 63.

MACRO P0 P1 ... Pn

sirve para definir una etiqueta como macro, así como para determinar su lista formal de parámetros; describiremos las macros más adelante.

END

indica el final del programa; el ensamblador ignorará todas las sentencias escritas a continuación.

ENDM

señala el fin de la definición de una macro.

ORDENES DEL ENSAMBLADOR

Las *órdenes* se usan para modificar el formato del listado y controlar la impresión del mismo. Todos empiezan con un asterisco en la primera columna. El ensamblador del Z80 dispone de siete, de las que las más típicas son:

*EJECT

que hace que el listado pase a la parte superior de la página siguiente: y

*LIST OFF

que determina la interrupción de la impresión. Las otras cinco son: "*HEADING S", "*LIST ON", "*MACLIST ON", "*MACLIST OFF", "*INCLUDE FILENAME".

MACROS

Una macro —o macroinstrucción— no es sino un grupo de varias instrucciones. Resulta muy cómodo para el programador poder sustituir una serie de instrucciones, que se utiliza muchas veces de la misma forma, por una macro que las represente, ya que se ahorra el trabajo de escribirlas todas las veces. Así, el grupo

```
SAVREG    MACRO
          PUSH AF
          PUSH BC
          PUSH DE
          PUSH HL
          ENDM
```

puede sustituirse a partir de este momento por "SAVREG". Cada vez que en el programa aparece SAVREG se ejecutan las cinco instrucciones a las que representa. Los ensambladores que tienen esta posibilidad se llaman macroensambladores, y se limitan a hacer una mera sustitución física de líneas equivalentes cada vez que encuentran el nombre de la macro.

¿Macro o subrutina?

A primera vista parece que una macro funciona como una subrutina, pero no es así. Cuando se utiliza el programa ensamblador para obtener el código objeto en el listado, la macroinstrucción se reemplaza por las instrucciones reales que la componen. Durante la ejecución, el grupo completo se repetirá tantas veces como su nombre genérico.

Por el contrario, la subrutina se define una sola vez, y el programa salta a su dirección cada vez que se utiliza. La macro es un recurso de *tiempo de ensamblado*, mientras que la subrutina lo es de *tiempo de ejecución*. Su comportamiento es muy diferente.

Parámetros macro

Cada macro puede equiparse con una serie de parámetros. Veamos un ejemplo:

```
SWAP    MACRO   #M, #N, #T
        LD      A, #M           ;M EN A
        LD      #T, A           ;A EN T (= M)
        LD      A, #N           ;N EN A
        LD      #M, A           ;A EN M (= N)
        LD      A, #T           ;T EN A
        LD      #N, A           ;A EN N (= T)
        END     M
```

Esta macroinstrucción intercambia los contenidos de las posiciones de memoria M y N, una operación con la que no cuenta el Z80, y que puede realizarse con una macro. En este caso, "T" es simplemente el nombre de una posición de almacenamiento temporal que necesita el programa. Vamos, por ejemplo, a intercambiar los contenidos de las posiciones de memoria ALFA y BETA:

```
SWAP (ALFA), (BETA), (TEMP)
```

En esta instrucción, TEMP es el nombre de una posición temporal de almacenamiento que sabemos que está disponible y que puede usar la macro. La descomposición de ésta en sus componentes elementales sería:

```
LD   A,(ALFA)
LD   (TEMP),A
LD   A,(BETA)
LD   (ALFA),A
LD   A,(TEMP)
LD   (BETA),A
```

Como es fácil comprobar, para el programador resulta útil utilizar seudoinstrucciones definidas con macros, porque ello le permite ampliar en apariencia las instrucciones del Z80. No obstante, hay que tener en cuenta que la ejecución se hace instrucción por instrucción, de modo que las macros funcionan más lentamente que las instrucciones aisladas. De todos modos, son muy útiles para desarrollar programas largos.

Otros recursos macro

A la simple posibilidad de crear macroinstrucciones se le pueden añadir recursos sintácticos y seudoinstrucciones adicionales, que permiten, por ejemplo, hacer macros *incluidas*, es decir, contenidas en una llamada macro más general. Si dentro de ésta se incluye una definición que modifique a la propia macro, ésta producirá en una primera llamada una expansión, y otra diferente en llamadas posteriores. El ensamblador del Z80 dispone de esta posibilidad, pero no de la de incluir definiciones dentro de otras.

Ensamblador condicional

Es otro de los recursos que proporciona el Z80. Gracias a él, el programador puede diseñar programas para casos muy diversos y ensamblar condicionalmente sus segmentos según el caso. Por ejemplo: un usuario crea un programa para controlar los semáforos de un cruce con diversos algoritmos de mando. A continuación recibe las instrucciones del ingeniero de tráfico local, en las que se especifican los semáforos y los algoritmos que deben emplearse. El programador se limita a fijar los parámetros en el programa general y ensamblarlo condicionalmente, operación que dará lugar a un programa "a la medida", que conservará únicamente las rutinas necesarias para resolver el problema específico que se le plantea.

Este recurso es útil en medios industriales, que cuentan siempre con varias opciones y que hacen interesante para el programador la posibilidad de montar rápida y automáticamente segmentos de programas en respuesta a los parámetros externos.

La versión del microensamblador del Z80 que proporciona Zilog sólo cuenta con dos seudooperaciones condicionales:

COND NN y ENDC

siendo NN una expresión. La seudooperación "COND NN" determina la evaluación de NN: si el resultado es positivo (no 0), se incluye la instrucción que sigue a COND; pero, si es nulo, se eliminan todas las instrucciones hasta la aparición de ENDC.

ENDC señala el final de COND y permite el ensamblado de todas las instrucciones que siguen. Estas seudooperaciones no pueden incluirse dentro de sí mismas.

Resumen

En teoría podría haber más posibilidades condicionales con las especificaciones "IF" y "ELSE"; quizá estén presentes en futuras versiones del ensamblador.

Resumen

Hemos visto en este capítulo las técnicas y los recursos físicos y lógicos que forman parte del desarrollo de los programas junto con diversas equivalencias y alternativas.

Estas van desde el microordenador monoplaca hasta un auténtico sistema de desarrollo, por lo que respecta al soporte físico, y desde el código binario hasta los lenguajes de programación de alto nivel, por lo que respecta al lógico. El lector deberá hacer la elección que considere óptima en función de sus recursos e intereses.



Desarrollo

11
Conclusión

Hemos recorrido todos los aspectos importantes de la programación, desde las definiciones y conceptos básicos hasta la manipulación interna de los registros del Z80, pasando por el control de los dispositivos de entrada/salida y los recursos lógicos de desarrollo. ¿Cuál es el siguiente paso? Hay dos perspectivas, una relacionada con el desarrollo de la tecnología, y la otra con el desarrollo de los conocimientos y la experiencia del propio programador. Examinémoslas brevemente.

Desarrollo tecnológico

El avance de la integración con tecnología MOS permite fabricar pastillas cada vez más complicadas. Paralelamente, el precio de producir el procesador propiamente dicho no deja de reducirse, de manera que en la actualidad casi todas las pastillas de entrada/salida y de control de periféricos montan procesadores sencillos y son *programables*, lo que plantea un dilema interesante: para simplificar el desarrollo de programas y para reducir el número de componentes, las nuevas pastillas de E/S disponen de refinados recursos programables que integran muchos algoritmos; en consecuencia, el desarrollo de programas

viene a complicarse, en contra de lo esperado, porque el programador debe estudiar en detalle todas estas nuevas pastillas. *Programar un sistema ya no es programar el microprocesador únicamente, sino también todas las pastillas conectadas a él*, y el tiempo necesario para llegar a dominar cada una de ellas puede ser considerable.

Pero el dilema es sólo aparente, porque, si no existiesen tales pastillas, la complejidad de las conexiones por realizar y de los correspondientes programas sería todavía mayor. La nueva complicación radica en la necesidad de programar más de un solo procesador y de estudiar las peculiaridades de cada una de las pastillas que componen el sistema. No obstante, cabe esperar que las técnicas y conceptos expuestos en este volumen hagan la tarea razonablemente fácil.

El siguiente paso

Ya se han estudiado las técnicas necesarias para programar aplicaciones sencillas, que era el objetivo de este libro. El paso siguiente es practicar, algo que no puede reemplazarse por ningún texto. Es imposible aprender a programar sólo estudiando: hay que practicar y adquirir experiencia. Ahora está en situación de escribir sus propios programas, y espero que esa nueva andadura le sea propicia.

APENDICE A

TABLA DE CONVERSION HEXADECIMAL

HEX	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	00	000
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	0
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	256	4096
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	512	8192
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	768	12288
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	1024	16384
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	1280	20480
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	1536	24576
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	1792	28672
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	2048	32768
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	2304	36864
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	2560	40960
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	2816	45056
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	3072	49152
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	3328	53248
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	3584	57344
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	3840	61440

5		4		3		2		1		0	
HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
0	0	0	0	0	0	0	0	0	0	0	0
1	1,048.576	1	65,536	1	4,096	1	256	1	16	1	1
2	2,097,152	2	131,072	2	8,192	2	512	2	32	2	2
3	3,145,728	3	196,608	3	12.288	3	768	3	48	3	3
4	4,194,304	4	262,144	4	16,384	4	1,024	4	64	4	4
5	5,242,880	5	327,680	5	20.480	5	1.280	5	80	5	5
6	6,291,456	6	393 216	6	24,576	6	1,536	6	96	6	6
7	7,340,032	7	458,752	7	28.672	7	1,792	7	112	7	7
8	8,388.608	8	524.288	8	32,768	8	2.048	8	128	8	8
9	9,437.184	9	589.824	9	36,864	9	2,304	9	144	9	9
A	10,485.760	A	655,360	A	40.960	A	2,560	A	160	A	10
B	11,534,336	B	720.896	B	45,056	B	2,816	B	176	B	11
C	12,582,912	C	786,432	C	49,152	C	3,072	C	192	C	12
D	13,631,488	D	851.968	D	53,248	D	3,328	D	208	D	13
E	14,680,064	E	917.504	E	57,344	E	3,584	E	224	E	14
F	15.728,640	F	983.040	F	61,440	F	3,840	F	240	F	15

APENDICE B

TABLA DE CONVERSION ASCII

HEX	MSD	0	1	2	3	4	5	6	7
LSD	BITS	000	001	010	011	100	101	110	111
0	0000	NUL	DLE	SPACE	0	@	P	–	p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	[illegible]	U	e	u
[illegible]	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	'	7	G	W	g	w
[illegible]	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	;	K	[	k	{
C	1100	FF	FS	,	<	L	\	l	- -
[illegible]	1101	CR	GS	–	=	M	]	[illegible]	}
E	1110	SO	RS	.	>	N	Λ	n	~
F	1111	SI	US	/	?	O	←	[illegible]	DEL

SIMBOLOS ASCII

NUL — Nulo
SOH — Principio [illegible] encabezamiento
STX — Principio de texto
ETX — Final de texto
ENQ — Pregunta
ACK — Reconocimiento
BEL — Timbre
BS — Retroceso
HT — Tabulación horizontal
LF — Alimentación de línea
VT — Tabulación vertical
FF — Alimentación de formato
CR — Retorno de carro
SO — Desplazamiento hacia afuera
[illegible] — Desplazamiento hacia adentro

DLE — Cambio de enlace de transmisión
DC — Control de dispositivo
NAK — Reconocimiento negativo
SYN — Funcionamiento síncrono
ETB — Fin de transmisión de bloque
CAN — Cancelar
EM — Fin de medio
SUB — Sustituir
ESC — Cambiar
[illegible] — Separador de ficheros
GS — Separador de grupos
RS — Separador de registros
US — Separador de unidades
SP — Espacio (blanco)
DEL — Borrar

APENDICE C

TABLAS DE BIFURCACION RELATIVA

TABLA DE BIFURCACION RELATIVA HACIA ADELANTE

MSD \ LSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127

TABLA DE BIFURCACION RELATIVA HACIA ATRAS

MSD \ LSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	128	127	126	125	124	123	122	121	120	119	118	117	116	115	114	113
9	112	111	110	109	108	107	106	105	104	103	102	101	100	99	98	97
A	96	95	94	93	92	91	90	89	88	87	86	85	84	83	82	81
B	80	79	78	77	76	75	74	73	72	71	70	69	68	67	66	65
C	64	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49
D	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33
E	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17
F	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1

APENDICE D

CONVERSION DECIMAL A BCD

DECIMAL	BCD	DEC	BCD	DEC	BCD
0	0000	10	00010000	90	10010000
1	0001	11	00010001	91	10010001
2	0010	12	00010010	92	10010010
3	0011	13	00010011	93	10010011
4	0100	14	00010100	94	10010100
5	0101	15	00010101	95	10010101
6	0110	16	00010110	96	10010110
7	0111	17	00010111	97	10010111
8	1000	18	00011000	98	10011000
9	1001	19	00011001	99	10011001

APENDICE E

CODIGOS DE LAS INSTRUCCIONES DEL Z80

(la letra d equivale en el código objeto a 05)

CODIGO OBJETO	INSTRUCCION FUENTE	
8E	ADC	A,(HL)
DD8E05	ADC	A,(IX+d)
FD8E05	ADC	A,(IY+d)
8F	ADC	A,A
88	ADC	A,B
89	ADC	A,C
8A	ADC	A,D
8B	ADC	A,E
8C	ADC	A,H
8D	ADC	A,L
CE20	ADC	A,n
ED4A	ADC	HL,BC
ED5A	ADC	HL,DE
ED6A	ADC	HL,HL
ED7A	ADC	HL,SP
86	ADD	A,(HL)
DD8605	ADD	A,(IX+d)
FD8605	ADD	A,(IY+d)
87	ADD	A,A
80	ADD	A,B
[illegible]	ADD	A,C
82	ADD	A,D
83	ADD	A,E
84	ADD	A,H
85	ADD	A,L
C620	ADD	A,n
09	ADD	HL,BC
19	ADD	HL,DE
29	ADD	HL,HL
39	ADD	HL,SP
DD09	ADD	IX,BC
DD19	ADD	IX,DE
DD29	ADD	IX,IX
DD39	ADD	IX,SP
FD09	ADD	IY,BC
FD19	ADD	IY,DE
FD29	ADD	IY,IY
FD39	ADD	IY,SP
A6	AND	(HL)
DDA605	AND	(IX+d)
FDA605	AND	(IY+d)
A7	AND	A
A0	AND	B
A1	AND	C
A2	AND	D
A3	AND	E
A4	AND	H
A5	AND	L

CODIGO OBJETO	INSTRUCCION FUENTE	
E620	AND	n
CB46	BIT	0,(HL)
DDCB0546	BIT	0,(IX+d)
FDCB0546	BIT	0,(IY+d)
CB47	BIT	0,A
CB40	BIT	0,B
CB41	BIT	0,C
CB42	BIT	0,D
CB43	BIT	0,E
CB44	BIT	0,H
CB45	BIT	0,L
CB4E	BIT	1,(HL)
DDCB054E	BIT	1,(IX+d)
FDCB054E	BIT	1,(IY+d)
CB4F	BIT	1,A
CB48	BIT	1,B
CB49	BIT	1,C
CB4A	BIT	1,D
CB4B	BIT	1,E
CB4C	BIT	1,H
CB4D	BIT	1,L
CB56	BIT	2,(HL)
DDCB0556	BIT	2,(IX+d)
FDCB0556	BIT	2,(IY+d)
CB57	BIT	2,A
CB50	BIT	2,B
CB51	BIT	2,C
CB52	BIT	2,D
CB53	BIT	2,E
CB54	BIT	2,H
CB55	BIT	2,L
CB5E	BIT	3,(HL)
DDCB055E	BIT	3,(IX+d)
FDCB055E	BIT	3,(IY+d)
CB5F	BIT	3,A
CB58	BIT	3,B
CB59	BIT	3,C
CB5A	BIT	3,D
CB5B	BIT	3,E
CB5C	BIT	3,H
CB5D	BIT	3,L
CB66	BIT	4,(HL)
DDCB0566	BIT	4,(IX+d)
FDCB0566	BIT	4,(IY+d)
CB67	BIT	4,A
CB60	BIT	4,B
CB61	BIT	4,C
CB62	BIT	4,D

CODIGO OBJETO	INSTRUCCION FUENTE	
CB63	BIT	4,E
CB64	BIT	4,H
CB65	BIT	4,L
CB6E	BIT	5,(HL)
DDCB056E	BIT	5,(IX+d)
FDCB056E	BIT	5,(IY+d)
CB6F	BIT	5,A
CB68	BIT	5,B
CB69	BIT	5,C
CB6A	BIT	5,D
CB6B	BIT	5,E
CB6C	BIT	5,H
CB6D	BIT	5,L
CB76	BIT	6,(HL)
DDCB0576	BIT	6,(IX+d)
FDCB0576	BIT	6,(IY+d)
CB77	BIT	6,A
CB70	BIT	6,B
CB71	BIT	6,C
CB72	BIT	6,D
CB73	BIT	6,E
CB74	BIT	6,H
CB75	BIT	6,L
CB7E	BIT	7,(HL)
DDCB057E	BIT	7,(IX+d)
FDCB057E	BIT	7,(IY+d)
CB7F	BIT	7,A
CB78	BIT	7,B
CB79	BIT	7,C
CB7A	BIT	7,D
CB7B	BIT	7,E
CB7C	BIT	7,H
CB7D	BIT	7,L
DC8405	CALL	C,nn
FC8405	CALL	M,nn
D48405	CALL	NC,nn
C48405	CALL	NZ,nn
F48405	CALL	P,nn
EC8405	CALL	PE,nn
E48405	CALL	PO,nn
CC8405	CALL	Z,nn
CD8405	CALL	nn
3F	CCF	
BE	CP	(HL)
DDBE05	CP	(IX+d)
FDBE05	CP	(IY+d)
BF	CP	A
B8	CP	B
B9	CP	C
BA	CP	D
BB	CP	E
BC	CP	H
BD	CP	L
FE20	CP	n
EDA9	CPD	
EDB9	CPDR	

CODIGO OBJETO	INSTRUCCION FUENTE	
EDB1	CPIR	
EDA1	CPI	
2F	CPL	
27	DAA	
35	DEC	(HL)
DD3505	DEC	(IX+d)
FD3505	DEC	(IY+d)
3D	DEC	A
05	DEC	B
0B	DEC	BC
0D	DEC	C
15	DEC	D
1B	DEC	DE
1D	DEC	E
25	DEC	H
2B	DEC	HL
DD2B	DEC	IX
FD2B	DEC	IY
2D	DEC	L
3B	DEC	SP
F3	DI	
102E	DJNZ	e
FB	EI	
E3	EX	(SP),HL
DDE3	EX	(SP),IX
FDE3	EX	(SP),IY
08	EX	AF,AF'
EB	EX	DE,HL
D9	EXX	
76	HALT	
ED46	IM	0
ED56	IM	1
ED5E	IM	2
ED78	IN	A,(C)
ED40	IN	B,(C)
ED48	IN	C,(C)
ED50	IN	D,(C)
ED58	IN	E,(C)
ED60	IN	H,(C)
ED68	IN	L,(C)
34	INC	(HL)
DD3405	INC	(IX+d)
FD3405	INC	(IY+d)
3C	INC	A
04	INC	B
03	INC	BC
0C	INC	C
14	INC	D
13	INC	DE
1C	INC	E
24	INC	H
23	INC	HL
DD23	INC	IX
FD23	INC	IY
2C	INC	L
33	INC	SP
DB20	IN	A,(n)

GO TO	INSTRUCCION FUENTE	
	CPIR	
	CPI	
	CPL	
	DAA	
	DEC	(HL)
5	DEC	(IX+d)
5	DEC	(IY+d)
	DEC	A
	DEC	B
	DEC	BC
	DEC	C
	DEC	D
	DEC	DE
	DEC	E
	DEC	H
	DEC	HL
	DEC	IX
	DEC	IY
	DEC	L
	DEC	SP
	DI	
	DJNZ	e
	EI	
	EX	(SP),HL
	EX	(SP),IX
	EX	(SP),IY
	EX	AF,AF'
	EX	DE,HL
	EXX	
	HALT	
	IM	0
	IM	1
	IM	2
	IN	A,(C)
	IN	B,(C)
	IN	C,(C)
	IN	D,(C)
	IN	E,(C)
	IN	H,(C)
	IN	L,(C)
	INC	(HL)
	INC	(IX+d)
	INC	(IY+d)
	INC	A
	INC	B
	INC	BC
	INC	C
	INC	D
	INC	DE
	INC	E
	INC	H
	INC	HL
	INC	IX
	INC	IY
	INC	L
	INC	SP
	IN	A,(n)

CODIGO OBJETO	INSTRUCCION FUENTE	
EDAA	IND	
EDBA	INDR	
EDA2	INI	
EDB2	INIR	
C38405	JP	nn
E9	JP	(HL)
DDE9	JP	(IX)
FDE9	JP	(IY)
DA8405	JP	C,nn
FA8405	JP	M,nn
D28405	JP	NC,nn
C28405	JP	NZ,nn
F28405	JP	P,nn
EA8405	JP	PE,nn
E28405	JP	PO,nn
CA8405	JP	Z,nn
382E	JR	C,e
302E	JR	NC,e
202E	JR	NZ,e
282E	JR	Z,e
182E	JR	e
02	LD	(BC),A
12	LD	(DE),A
77	LD	(HL),A
70	LD	(HL),B
71	LD	(HL),C
72	LD	(HL),D
73	LD	(HL),E
74	LD	(HL),H
75	LD	(HL),L
3620	LD	(HL),n
DD7705	LD	(IX+d),A
DD7005	LD	(IX+d),B
DD7105	LD	(IX+d),C
DD7205	LD	(IX+d),D
DD7305	LD	(IX+d),E
DD7405	LD	(IX+d),H
DD7505	LD	(IX+d),L
DD360520	LD	(IX+d),n
FD7705	LD	(IY+d),A
FD7005	LD	(IY+d),B
FD7105	LD	(IY+d),C
FD7205	LD	(IY+d),D
FD7305	LD	(IY+d),E
FD7405	LD	(IY+d),H
FD7505	LD	(IY+d),L
FD360520	LD	(IY+d),n
328405	LD	(nn),A
ED438405	LD	(nn),BC
ED538405	LD	(nn),DE
228405	LD	(nn),HL
DD228405	LD	(nn),IX
FD228405	LD	(nn),IY
ED738405	LD	(nn),SP
0A	LD	A,(BC)
1A	LD	A,(DE)
7E	LD	A,(HL)

CODIGO OBJETO	INSTRUCCION FUENTE	
DD7E05	LD	A,(IX+d)
FD7E05	LD	A,(IY+d)
3A8405	LD	A,(nn)
7F	LD	A,A
78	LD	A,B
79	LD	A,C
7A	LD	A,D
7B	LD	A,E
7C	LD	A,H
ED57	LD	A,I
7D	LD	A,L
3E20	LD	A,n
ED5F	LD	A,R
46	LD	B,(HL)
DD4605	LD	B,(IX+d)
FD4605	LD	B,(IY+d)
47	LD	B,A
40	LD	B,B
41	LD	B,C
42	LD	B,D
43	LD	B,E
44	LD	B,H
45	LD	B,L
0620	LD	B,n
ED4B8405	LD	BC,(nn)
018405	LD	BC,nn
4E	LD	C,(HL)
DD4E05	LD	C,(IX+d)
FD4E05	LD	C,(IY+d)
4F	LD	C,A
48	LD	C,B
49	LD	C,C
4A	LD	C,D
4B	LD	C,E
4C	LD	C,H
4D	LD	C,L
0E20	LD	C,n
56	LD	D,(HL)
DD5605	LD	D,(IX+d)
FD5605	LD	D,(IY+d)
57	LD	D,A
50	LD	D,B
51	LD	D,C
52	LD	D,D
53	LD	D,E
54	LD	D,H
55	LD	D,L
1620	LD	D,n
ED5B8405	LD	DE,(nn)
118405	LD	DE,nn
5E	LD	E,(HL)
DD5E05	LD	E,(IX+d)
FD5E05	LD	E,(IY+d)
5F	LD	E,A
58	LD	E,B
58	LD	E,C
5A	LD	E,D

CODIGO OBJETO	INSTRUCCION FUENTE	
5B	LD	E,E
5C	LD	E,H
5D	LD	E,L
1E20	LD	E,n
66	LD	H,(HL)
DD6605	LD	H,(IX+d)
FD6605	LD	H,(IY+d)
67	LD	H,A
[illegible]	LD	H,B
61	LD	H,C
62	LD	H,D
63	LD	H,E
64	LD	H,H
65	LD	H,L
2620	LD	H,n
2A8405	LD	HL,(nn)
218405	LD	HL,nn
ED47	LD	I,A
DD2A8405	LD	IX,(nn)
DD218405	LD	IX,nn
FD2A8405	LD	IY,(nn)
FD218405	LD	IY,nn
6E	LD	L,(HL)
DD6E05	LD	L,(IX+d)
FD6E05	LD	L,(IY+d)
6F	LD	L,A
[illegible]	LD	L,B
69	LD	L,C
6A	LD	L,D
6B	LD	L,E
6C	LD	L,H
6D	LD	L,L
2E20	LD	L,n
ED4F	LD	R,A
ED7B8405	LD	SP,(nn)
F9	LD	SP,HL
DDF9	LD	SP,IX
FDF9	LD	SP,IY
318405	LD	SP,nn
EDA8	LDD	
EDB8	LDDR	
EDA0	LDI	
EDB0	LDIR	
ED44	NEG	
00	NOP	
B6	OR	(HL)
DDB605	OR	(IX+d)
FDB605	OR	(IY+d)
B7	OR	A
B0	OR	B
B1	OR	C
B2	OR	D
B3	OR	E
B4	OR	H
B5	OR	L
F620	OR	n
EDBB	OTDR	

CODIGO OBJETO	INSTRUCCION FUENTE	
EDB3	OTIR	
ED79	OUT	(C),A
ED41	OUT	(C),B
ED49	OUT	(C),C
ED51	OUT	(C),D
ED59	OUT	(C),E
ED61	OUT	(C),H
ED69	OUT	(C),L
D320	OUT	(n),A
EDAB	OUTD	
EDA3	OUTI	
F1	POP	AF
C1	POP	BC
D1	POP	DE
E1	POP	HL
DDE1	POP	IX
FDE1	POP	IY
F5	PUSH	AF
C5	PUSH	BC
D5	PUSH	DE
[illegible]	PUSH	HL
DDE5	PUSH	IX
FDE5	PUSH	IY
CB86	RES	0,(HL)
DDCB0586	RES	0,(IX+d)
FDCB0586	RES	0,(IY+d)
CB87	RES	0,A
CB80	RES	0,B
CB81	RES	0,C
CB82	RES	0,D
CB83	RES	0,E
CB84	RES	0,H
CB85	RES	0,L
CB8E	RES	1,(HL)
DDCB058E	RES	1,(IX+d)
FDCB058E	RES	1,(IY+d)
CB8F	RES	1,A
CB88	RES	1,B
CB89	RES	1,C
CB8A	RES	1,D
CB8B	RES	1,E
CB8C	RES	1,H
CB8D	RES	1,L
CB96	RES	2,(HL)
DDCB0596	RES	2,(IX+d)
FDCB0596	RES	2,(IY+d)
CB97	RES	2,A
CB90	RES	2,B
CB91	RES	2,C
CB92	RES	2,D
CB93	RES	2,E
CB94	RES	2,H
CB95	RES	2,L
CB9E	RES	3,(HL)
DDCB059E	RES	3,(IX+d)
FDCB059E	RES	3,(IY+d)

O	INSTRUCCION FUENTE	
	OTIR	
	OUT	(C),A
	OUT	(C),B
	OUT	(C),C
	OUT	(C),D
	OUT	(C),E
	OUT	(C),H
	OUT	(C),L
	OUT	(n),A
	OUTD	
	OUTI	
	POP	AF
	POP	BC
	POP	DE
	POP	HL
	POP	IX
	POP	IY
	PUSH	AF
	PUSH	BC
	PUSH	DE
	PUSH	HL
	PUSH	IX
	PUSH	IY
	RES	0,(HL)
-86	RES	0,(IX+d)
86	RES	0,(IY+d)
	RES	0,A
	RES	0,B
	RES	0,C
	RES	0,D
	RES	0,E
	RES	0,H
	RES	0,L
	RES	1,(HL)
BE	RES	1,(IX+d)
IE	RES	1,(IY+d)
	RES	1,A
	RES	1,B
	RES	1,C
	RES	1,D
	RES	1,E
	RES	1,H
	RES	1,L
	RES	2,(HL)
[redacted]	RES	2,(IX+d)
6	RES	2,(IY+d)
	RES	2,A
	RES	2,B
	RES	2,C
	RES	2,D
	RES	2,E
	RES	2,H
	RES	2,L
	RES	3,(HL)
E	RES	3,(IX+d)
E	RES	3,(IY+d)

CODIGO OBJETO	INSTRUCCION FUENTE	
CB9F	RES	3,A
CB98	RES	3,B
CB99	RES	3,C
CB9A	RES	3,D
CB9B	RES	3,E
CB9C	RES	3,H
CB9D	RES	3,L
CBA6	RES	4,(HL)
DDCB05A6	RES	4,(IX+d)
FDCB05A6	RES	4,(IY+d)
CBA7	RES	4,A
CBA0	RES	4,B
CBA1	RES	4,C
CBA2	RES	4,D
CBA3	RES	4,E
CBA4	RES	4,H
CBA5	RES	4,L
CBAE	RES	5,(HL)
DDCB05AE	RES	5,(IX+d)
FDCB05AE	RES	5,(IY+d)
CBAF	RES	5,A
CBA8	RES	5,B
CBA9	RES	5,C
CBAA	RES	5,D
CBAB	RES	5,E
CBAC	RES	5,H
CBAD	RES	5,L
CBB6	RES	6,(HL)
DDCB05B6	RES	6,(IX+d)
FDCB05B6	RES	6,(IY+d)
CBB7	RES	6,A
CBB0	RES	6,B
CBB1	RES	6,C
CBB2	RES	6,D
CBB3	RES	6,E
CBB4	RES	6,H
CBB5	RES	6,L
CBBE	RES	7,(HL)
DDCB05BE	RES	7,(IX+d)
FDCB05BE	RES	7,(IY+d)
CBBF	RES	7,A
CBB8	RES	7,B
CBB9	RES	7,C
CBBA	RES	7,D
CBBB	RES	7,E
CBBC	RES	7,H
CBBD	RES	7,L
C9	RET	
D8	RET	C
F8	RET	M
D0	RET	NC
C0	RET	NZ
F0	RET	P
[redacted]	RET	PE
[redacted]	RET	PO
C8	RET	Z

CODIGO OBJETO	INSTRUCCION FUENTE	
ED4D	RETI	
ED45	RETN	
CB16	RL	(HL)
DDCB0516	RL	(IX+d)
FDCB0516	RL	(IY+d)
CB17	RL	A
CB10	RL	B
CB11	RL	C
CB12	RL	D
CB13	RL	E
CB14	RL	H
CB15	RL	L
17	RLA	
CB06	RLC	(HL)
DDCB0506	RLC	(IX+d)
FDCB0506	RLC	(IY+d)
CB07	RLC	A
CB00	RLC	[redacted]
CB01	RLC	C
CB02	RLC	[redacted]
CB03	RLC	E
CB04	RLC	[redacted]
CB05	RLC	L
[redacted]	RLCA	
ED6F	RLD	
CB1E	RR	(HL)
DDCB051E	RR	(IX+d)
FDCB051E	RR	(IY+d)
CB1F	RR	A
CB18	RR	B
CB19	RR	[redacted]
CB1A	RR	[redacted]
CB1B	RR	E
CB1C	RR	H
CB1D	RR	L
1F	RRA	
CB0E	RRC	(HL)
DDCB050E	RRC	(IX+d)
FDCB050E	RRC	(IY+d)
CB0F	RRC	A
CB08	RRC	[redacted]
CB09	RRC	C
CB0A	RRC	D
CB0B	RRC	E
CB0C	RRC	H
CB0D	RRC	L
0F	RRCA	
ED67	RRD	
C7	RST	00H
CF	RST	08H
D7	RST	10H
DF	RST	18H
E7	RST	20H
EF	RST	28H
F7	RST	30H
FF	RST	38H
DE20	SBC	A,n

CODIGO OBJETO	INSTRUCCION FUENTE	
9E	SBC	A,(HL)
DD9E05	SBC	A,(IX+d)
FD9E05	SBC	A,(IY+d)
9F	SBC	A,A
98	SBC	A,B
99	SBC	A,C
9A	SBC	A,D
9B	SBC	A,E
9C	SBC	A,H
9D	SBC	A,L
ED42	SBC	HL,BC
ED52	SBC	HL,DE
ED62	SBC	HL,HL
ED72	SBC	HL,SP
37	SCF	
CBC6	SET	0,(HL)
DDCB05C6	SET	0,(IX+d)
FDCB05C6	SET	0,(IY+d)
CBC7	SET	0,A
CBC0	SET	0,B
CBC1	SET	0,C
CBC2	SET	0,D
CBC3	SET	0,E
CBC4	SET	0,H
CBC5	SET	0,L
CBCE	SET	1,(HL)
DDCB05CE	SET	1,(IX+d)
FDCB05CE	SET	1,(IY+d)
CBCF	SET	1,A
CBC8	SET	1,B
CBC9	SET	1,C
CBCA	SET	1,D
CBCB	SET	1,E
CBCC	SET	1,H
CBCD	SET	1,L
CBD6	SET	2,(HL)
DDCB05D6	SET	2,(IX+d)
FDCB05D6	SET	2,(IY+d)
CBD7	SET	2,A
CBD0	SET	2,B
CBD1	SET	2,C
CBD2	SET	2,D
CBD3	SET	2,E
CBD4	SET	2,H
CBD5	SET	2,L
CBD8	SET	3,B
CBDE	SET	3,(HL)
DDCB05DE	SET	3,(IX+d)
FDCB05DE	SET	3,(IY+d)
CBDF	SET	3,A
CBD9	SET	3,C
CBDA	SET	3,D
CBDB	SET	3,E
CBDC	SET	3,H
CBDD	SET	3,L
CBE6	SET	4,(HL)
DDCB05E6	SET	4,(IX+d)
FDCB05E6	SET	4,(IY+d)
CBE7	SET	4,A
CBE0	SET	4,B
CBE1	SET	4,C
CBE2	SET	4,D
CBE3	SET	4,E
CBE4	SET	4,H
CBE5	SET	4,L
CBEE	SET	5,(HL)
DDCB05EE	SET	5,(IX+d)
FDCB05EE	SET	5,(IY+d)
CBEF	SET	5,A
[illegible]	SET	5,B
CBE9	SET	5,C
CBEA	SET	5,D
CBEB	SET	5,E
CBEC	SET	5,H
CBED	SET	5,L
CBF6	SET	6,(HL)
DDCB05F6	SET	6,(IX+d)
FDCB05F6	SET	6,(IY+d)
CBF7	SET	6,A
CBF0	SET	6,B
CBF1	SET	6,C
CBF2	SET	6,D
CBF3	SET	6,E
CBF4	SET	6,H
CBF5	SET	6,L
CBFE	SET	7,(HL)
DDCB05FE	SET	7,(IX+d)
FDCB05FE	SET	7,(IY+d)
CBFF	SET	7,A
CBF8	SET	7,B
CBF9	SET	7,C
CBFA	SET	7,D
CBFB	SET	7,E
CBFC	SET	7,H
CBFD	SET	7,L
CB26	SLA	(HL)
DDCB0526	SLA	(IX+d)
FDCB0526	SLA	(IY+d)
CB27	SLA	A
CB20	SLA	B
CB21	SLA	C
CB22	SLA	D
CB23	SLA	E
CB24	SLA	H
CB25	SLA	L
CB2E	SRA	(HL)
DDCB052E	SRA	(IX+d)
FDCB052E	SRA	(IY+d)
CB2F	SRA	A
CB28	SRA	B
CB29	SRA	C
CB2A	SRA	D

CODIGO OBJETO	INSTRUCCION FUENTE	
CB2B	SRA	E
CB2C	SRA	H
CB2D	SRA	L
CB3E	SRL	(HL)
DDCB053E	SRL	(IX+d)
FDCB053E	SRL	(IY+d)
CB3F	SRL	A
CB38	SRL	B
CB39	SRL	C
CB3A	SRL	D
CB3B	SRL	E
CB3C	SRL	H
CB3D	SRL	L
96	SUB	(HL)
DD9605	SUB	(IX+d)
FD9605	SUB	(IY+d)
97	SUB	A
90	SUB	B
91	SUB	C
92	SUB	D
93	SUB	E
[illegible]	SUB	H
95	SUB	L
D620	SUB	n
AE	XOR	(HL)
DDAE05	XOR	(IX+d)
FDAE05	XOR	(IY+d)
AF	XOR	A
A8	XOR	B
A9	XOR	C
AA	XOR	D
AB	XOR	E
AC	XOR	H
AD	XOR	L
EE20	XOR	n

(Cortesía de Zilog Inc.)

APENDICE F

EQUIVALENCIAS DEL Z80 AL 8080

Z80	8080	Z80	8080	Z80	8080
ADC A, (HL)	ADC M	EX (SP), HL	XTHL	OR n	ORI [B2]
ADC A, n	ACI [B2]	HALT	HLT	OR r	ORA r
ADC A, r	ADC r	IN A, (n)	IN [B2]	OR (HL)	ORA M
ADD A, (HL)	ADD M	INC BC	INX B	OUT (n), A	OUT [B2]
ADD A, n	ADI [B2]	INC DE	INX D	POP AF	POP PSW
ADD A, r	ADD r	INC HL	INX H	POP BC	POP B
ADD HL, BC	DAD B	INC r	INR r	POP DE	POP D
ADD HL, DE	DAD D	INC SP	INX SP	POP HL	POP H
ADD HL, HL	DAD H	INC (HL)	INR M	PUSH AF	PUSH PSW
ADD HL, SP	DAD SP	JP C, nn	JC [B2] [B3]	PUSH BC	PUSH B
AND n	ANI [B2]	JP M, nn	JM [B2][B3]	PUSH DE	PUSH D
AND r	ANA r	JP NC, nn	JNC [B2] [B3]	PUSH HL	PUSH H
AND (HL)	ANA M	JP nn	JMP [B2] [B3]	RET	RET
CALL C, nn	CC [B2] [B3]	JP NZ, nn	JNZ [B2] [B3]	RET C	RC
CALL M, nn	CM [B2] [B3]	JP P, nn	JP [B2] [B3]	RET M	RM
CALL NC, nn	CNC [B2] [B3]	JP PE, nn	JPE [B2][B3]	RET NC	RNC
CALL nn	CALL	JP PO, nn	JPO [B2][B3]	RET NZ	RNZ
CALL NZ, nn	CNZ [B2] [B3]	JP Z, nn	JZ [B2] [B3]	RET P	RP
CALL P, nn	CP [B2] [B3]	JP (HL)	PCHL	RET PE	RPE
CALL PE, nn	CPE [B2] [B3]	LD A, (DE)	LDAX	RET PO	RPO
CALL PO, nn	CPO [B2] [B3]	LD A, (nn)	LDA [B2] [B3]	RET Z	RZ
CALL Z, nn	CZ [B2] [B3]	LD DE, nn	LXI D [B2] [B3]	RLA	RAL
CCF	CMC	LD SP, nn	LXI SP [B2] [B3]	RLCA	RLC
CP r	CMP r	LD (BC), A	STAX B	RRA	RAR
CP (HL)	CMP M	LD (DE), A	STAX D	RRCA	RRC
CPL	CMA	LD (HL), r	MOV M, r	RST P	RST P
CP n	CPI [B2]	LD (nn), A	STA [B2] [B3]	SBC A, (HL)	SBB M
DAA	DAA	LD (nn), HL	SHLD [B2] [B3]	SBC A, n	SBI [B2]
DEC BC	DCX B	LD A, (BC)	LDAX B	SBC A, r	SBB r
DEC DE	DCX D	LD BC, nn	LXI B, [B2] [B3]	SCF	STC
DEC HL	DCX H	LD HL, (nn)	LHLD [B2] [B3]	SUB n	SUI [B2]
DEC r	DCR r	LD HL, nn	LXI H [B2] [B3]	SUB r	SUB r
DEC SP	DCX SP	LD r, (HL)	MOV r, M	SUB (HL)	SUB M
DEC (HL)	DCR M	LD r, n	MVI r, [B2]	XOR n	XRI [B2]
DI	DI	LD r, r'	MOV r1, r2	XOR r	XRA r
EI	EI	LD SP, HL	SPHL	XOR (HL)	XRA M
EX DE, HL	XCHG	NOP	NOP		

APENDICE G

EQUIVALENCIAS DEL 8080 AL Z80

8080	Z80	8080	Z80	8080	Z80
ACI [B2]	ADC A, n	IN [B2]	IN A, (n)	POP H	POP HL
ADC M	ADC A, (HL)	INR M	INC (HL)	POP PSW	POP AF
ADC r	ADC A, r	INR r	INC r	PUSH B	PUSH BC
ADD M	ADD A, (HL)	INX B	INC BC	PUSH D	PUSH DE
ADD r	ADD A, r	INX D	INC DE	PUSH H	PUSH HL
ADI [B2]	ADD A, n	INX H	INC HL	PUSH PSW	PUSH AF
ANA M	AND (HL)	INX SP	INC SP	RAL	RLA
ANA r	AND r	JC [B2] [B3]	JP C, nn	RAR	RRA
ANI [B2]	AND n	JM [B2] [B3]	JP M, nn	RC	RET C
CALL	CALL nn	JMP [B2] [B3]	JP nn	RET	RET
CC [B2] [B3]	CALL C, nn	JNC [B2] [B3]	JP NC, nn	RLC	RLCA
CM [B2] [B3]	CALL M, nn	JNZ [B2] [B3]	JP NZ, nn	RM	RET M
CMA	CPL	JP [B2] [B3]	JP P, nn	RNC	RET NC
CMC	CCF	JPE [B2] [B3]	JP PE, nn	RNZ	RET NZ
CMP M	CP (HL)	JPO [B2] [B3]	JP PO, nn	RP	RET P
CMP r	CP r	JZ [B2] [B3]	JP Z, nn	RPE	RET PE
CNC [B2] [B3]	CALL NC, nn	LDA [B2] [B3]	LD A, (nn)	RPO	RET PO
CNZ [B2] [B3]	CALL NZ, nn	LDAX B	LD A, (BC)	RRC	RRCA
CP [B2] [B3]	CALL P, nn	LDAX D	LD A, (DE)	RST	RST P
CPE [B2] [B3]	CALL PE, nn	LHLD [B2] [B3]	LD HL, (nn)	RZ	RET Z
CPI [B2]	CP n	LXI B [B2] [B3]	LD BC, nn	SBB M	SBC A, (HL)
CPO [B2] [B3]	CALL PO, nn	LXI D [B2] [B3]	LD DE, nn	SBB r	SBC A, r
CZ [B2] [B3]	CALL Z, nn	LXI H [B2] [B3]	LD HL, nn	SBI [B2]	SBC A, n
DAA	DAA	LXI SP [B2] [B3]	LD SP, nn	SHLD [B2] [B3]	LD (nn), HL
DAD B	ADD HL, BC	MOV M, r	LD (HL), r	SPHL	LD SP, HL
DAD D	ADD HL, DE	MOV r, M	LD r, (HL)	STA [B2] [B3]	LD (nn), A
DAD H	ADD HL, HL	MOV r1, r2	LD r, r'	STAX B	LD (BC), A
DAD SP	ADD HL, SP	MVI M	LD (HL), n	STAX D	LD (DE), A
DCR M	DEC (HL)	MVI r [B2]	LD r, n	STC	SCF
DCR r	DEC r	NOP	NOP	SUB M	SUB (HL)
DCX B	DEC BC	ORA M	OR (HL)	SUB r	SUB r
DCX D	DEC DE	ORA r	OR r	SUI [B2]	SUB n
DCX H	DEC HL	ORI [B2]	OR n	XCHG	EX DE, HL
DCX SP	DEC SP	OUT [B2]	OUT (n), A	XRA M	XOR (HL)
DI	DI	PCHL	JP (HL)	XRA r	XOR r
EI	EI	POP B	POP BC	XRI [B2]	XOR n
HALT	HLT	POP D	POP DE	XTHL	EX (SP), HL

Indice alfabético

Programación del Z80

PROGRAMACION DEL Z80 va dirigido tanto a quien toma contacto por primera vez con el Z80 y desea conocer a fondo su funcionamiento, como para el programador experimentado que necesita una guía de referencia rápida, completa y concisa sobre todos los pormenores del Z80.

El libro contiene una descripción detallada del ***hardware*** *del procesador (registros, buses, etc.) y una extensa* ***guía de programación*** *que trata, de forma gradual y con numerosos ejemplos, todos los temas de programación en lenguaje máquina:*

- *modos de direccionamiento;*
- *técnicas complejas de entrada/salida;*
- *interrupciones;*
- *programas aritméticos, búsqueda, ordenación, etc.*

PROGRAMACION DEL Z80 incluye también un extenso capítulo con una descripción detallada del ***juego de instrucciones*** *del Z80: código operativo, función, flujo de datos, modo de direccionamiento, tiempo de ejecución, etc.*

Con más de 200 ilustraciones y siete apéndices, PROGRAMACION DEL Z80 es una obra de referencia imprescindible en la biblioteca de cualquier programador.



Programación del Z80 Rodnay Zaks

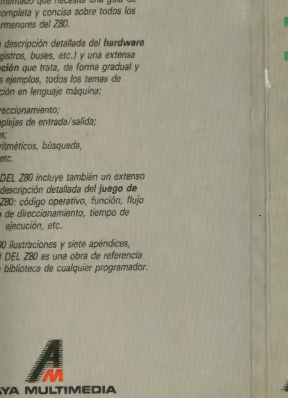
Programación del Z80

Rodnay Zaks



ramación

el Z80



Programación del Z80 Rodnay Zaks

Program
del Z

Rodnay Z

Programación del Z80

PROGRAMACION DEL Z80 va dirigido tanto a quien toma contacto por primera vez con el Z80 y desea conocer a fondo su funcionamiento, como para el programador experimentado que necesita una guía de referencia rápida, completa y concisa sobre todos los pormenores del Z80.

El libro contiene una descripción detallada del **hardware** del procesador (registros, buses, etc.) y una extensa **guía de programación** que trata, de forma gradual y con numerosos ejemplos, todos los temas de programación en lenguaje máquina:

- modos de direccionamiento;
- técnicas complejas de entrada/salida;
- interrupciones;
- programas aritméticos, búsqueda, ordenación, etc.

PROGRAMACION DEL Z80 incluye también un extenso capítulo con una descripción detallada del **juego de instrucciones** del Z80: código operativo, función, flujo de datos, modo de direccionamiento, tiempo de ejecución, etc.

Con más de 200 ilustraciones y siete apéndices, PROGRAMACION DEL Z80 es una obra de referencia imprescindible en la biblioteca de cualquier programador.



Programación del Z80 Rodnay Zaks